UNIVERSITY OF OSLO
Department of informatics

Modeling Activity Diagram on Eclipse

Master thesis
60 credits

Geoffrey Rekier

01. August 2008

Abstract

MADE is a tool for project management from activity diagram. It is based on Eclipse Graphic Modeling Framework (GMF) and Meta-Object Facility (MOF) Script. It can help you see what is not working as planned and help you improve your workflow. You can add constraints that can be checked by the tool and it will report if something went wrong or tell you that everything is right.

The purpose of this paper is to look into how we can use Activity Diagram for Project Planning. We will use Model Driven Development to achieve this.

Acknowledgements

This thesis is submitted to the Department of Informatics at the University of Oslo as part of a Master Degree. The work reported in this thesis has been started at the University of Oslo in the Modeling group and finished at SINTEF.

I would like to thank my supervisors Dr. Øystein Haugen for support and guidance beyond anything I could expect. He has shown patience over these two years and always has constructive comments to set me in the right direction even when he was on holiday.

I would also like to thank Andreas Limyr that gave me an introduction to the tools used here and Svein Melby for his help within MOFscript.

Finally, I need to thank my family for its support, specially my wife. You have all given me the encouragement and moral support I have needed to finish this thesis.

Table of Contents

Figures

1. Introduction

1.1. Motivations and Background

MADE started in 2006 as a project to show how we could generate code from Activity Diagram. We needed to create an Activity Diagram editor that we could transform from. This means that the editor should be connected to a model we could work from. This was often done with the Eclipse Modeling Framework (EMF)[1] and the Graphical Editing Framework (GEF)[2] like in SeDi. But we thought about the upcoming GMF as a tool for this because its purpose is to transform models to a graphical editor on the Eclipse Platform. It looked like the perfect tool for us. There were many projects that started using GMF to create UML editors like UML tool in Eclipse or Petri Nets modeling and it seemed like they were working. We started with a simple Class Diagram editor and it was quite easy.

After some simple tests, we tried to create an Activity Diagram editor and this did not go as well. GMF was not working as expected. There were many bugs, it generated code with errors and hanged a lot even crashed the whole Eclipse Platform. In short, it was young and not stable. We used a lot of time trying to make this work and, after every fix that was released also acquiring better knowledge of GMF, we made it. We created an Activity Diagram editor.

After that, we created a model-to-text transformation that would allow us to generate some code. We wanted to test this to achieve project management because we think that Activity Diagrams are good to represent business processes for a project manager with little knowledge of modeling. The purpose was restricted but we are able to show that we can generate code from Activity Diagrams and that we can use this code for project control.

1.2. UML Activity Diagram

UML is an OMG technology that is defined by the UML metamodel. This is the most used OMG specification, and the way the world models not only application structure, behavior, and architecture, but also business process and data structure. UML, along with the Meta Object Facility (MOF™), also provides a key foundation for OMG's Model-Driven Architecture, which tries to unify every step of development and integration from business modeling, through

architectural and application modeling, to development, deployment, maintenance, and evolution.

Activity[3] modeling emphasizes the sequence and conditions for coordinating lower-level behaviors, rather than which classifiers own those behaviors. These are commonly called control flow and object flow models. The actions coordinated by activity models can be initiated when other actions finish executing, because objects and data become available, or events occur external to the flow.

Activity diagrams are typically used for business process modeling, for modeling the logic captured by a single use case or usage scenario, or for modeling the detailed logic of a business rule. Although UML activity diagrams could potentially model the internal logic of a complex operation it would be far better to simply rewrite the operation in a way that it would be simple enough so you do not require an Activity Diagram. In many ways UML activity diagrams are the object-oriented equivalent of flow charts and data flow diagrams (DFDs) from structured development.[4]

The purpose of the activity diagram is to model the procedural flow of actions that are part of a larger activity. In projects in which use cases are present, activity diagrams can model a specific use case at a more detailed level. However, activity diagrams can be used independently of use cases for modeling a business-level function, such as buying a concert ticket or registering for a college class. Activity diagrams can also be used to model system-level functions, such as how a ticket reservation datamart populates a corporate sales system's data warehouse. Because it models procedural flow, the activity diagram focuses on the action sequence of execution and the conditions that trigger or guard those actions. The activity diagram is also focused only on the activity's internal actions and *not* on the actions that call the activity in their process flow or that trigger the activity according to some event (e.g., it is 12:30 on April 13th, and Green Day tickets are now on sale for the group's summer tour).

Although UML Sequence Diagrams can portray the same information as Activity Diagrams, we personally find activity diagrams best for modeling business-level functions. This is because Activity Diagrams show all potential sequence flows in an activity, whereas a Sequence Diagram typically shows only one flow of an activity. In addition, business managers and

business process personnel seem to prefer activity diagrams over sequence diagrams -- an activity diagram is less "techie" in appearance, and therefore less intimidating to business people. Besides, business managers are used to seeing flow diagrams, so the "look" of an activity diagram is familiar.

1.3. Eclipse Platform

Eclipse[5, 6] is an open source community whose projects are focused on building an open development platform comprised of extensible frameworks, tools and runtimes for building, deploying and managing software across the lifecycle. A large and vibrant ecosystem of major technology vendors, innovative start-ups, universities, research institutions and individuals extend, complement and support the Eclipse platform.

Eclipse began as an IBM Canada project. It was developed by OTI (Object Technology International) as a replacement for VisualAge, which itself had been developed by OTI. In November 2001, a consortium was formed to further the development of Eclipse as open source. In 2003, the Eclipse Foundation was created.

Eclipse 3.0 (released on June 21 2004) selected the OSGi Service Platform specifications as the runtime architecture. Eclipse was originally released under the Common Public License, but was later relicensed under the Eclipse Public License. The Free Software Foundation has said that both licenses are free software licenses, but are incompatible with the GNU General Public License (GPL). Mike Milinkovich, of the Eclipse Foundation has commented that moving to the GPL will be considered when version 3 of the GPL is released.

The Eclipse Platform subproject provides the core frameworks and services upon which all plug-in extensions are created. It also provides the runtime in which plug-ins are loaded, integrated, and executed. The primary purpose of the Platform subproject is to enable other tool developers to easily build and deliver integrated tools.

The Eclipse platform itself is a sort of universal tool platform - it is an IDE for anything and nothing in particular. It can deal with any type of resource (Java files, C files, Word files, HTML files, JSP files, etc) in a generic manner but does not know how to do anything that is specific to a particular file type. The Eclipse platform, by itself, does not provide a great deal of end-user functionality - it is what it enables that is interesting. The real value comes from

tool plug-ins for eclipse that "teaches" the platform how to work with these different kinds of resources. This pluggable architecture allows a more seamless experience for the end user when moving between different tools than ever before possible.

The Eclipse platform defines a set of frameworks and common services that collectively make up "integration-ware" required to support a comprehensive tool integration platform. These services and frameworks represent the common facilities required by most tool builders including a standard workbench user interface and project model for managing resources, portable native widget and user interface libraries, automatic resource delta management for incremental compilers and builders, language-independent debug infrastructure, and infrastructure for distributed multi-user versioned resource management.

In addition, the Eclipse platform defines a workbench user interface and a set of common domain-independent user interaction paradigms that tool builders plug into to add new capabilities. The platform comes with a set of standard views which can be extended by tool builders. Tool builders can both add new views, and plug new domain-specific capability into existing views.

When people speak of Eclipse, they very often mean the Eclipse Software Development Kit (SDK) which is both one of the leading Java™ integrated development environment (IDE) and the single best tool available for building products based on the Eclipse Platform. The Eclipse SDK, a critical piece of the Eclipse tapestry, is a combination of the efforts of several Eclipse projects, including Platform [http://eclipse.org/platform], Java Development Tools (JDT [http://eclipse.org/jdt]), and the Plug-in Development Environment (PDE [http://eclipse.org/pde]).

In its entirety, the Eclipse Platform contains the functionality required to build an IDE. However, the Eclipse Platform is itself a composition of components; by using a subset of these components, it is possible to build arbitrary applications. The Eclipse Rich Client Platform (RCP) is one such subset of components. Figure 1 shows a representation of some of the components in the Eclipse Platform and highlights the subset that makes up the RCP (in reality there are a great many more components).

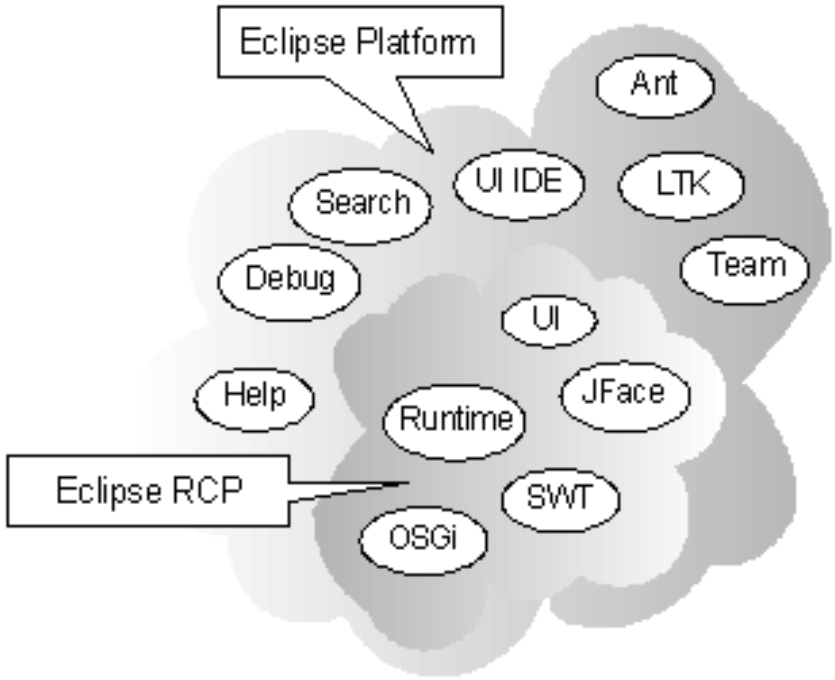

Figure 1-1 The Eclipse Rich Client Platform (RCP) is a subset of the Eclipse Platform.

Eclipse Platform is more than just a foundation for building development environments: it is a foundation for building arbitrary tools and applications. The RCP is being used to build arbitrary applications that have nothing to do with software development in diverse areas that include banking, automotive, medical, and space exploration. As the name "rich client" implies, Eclipse RCP is a good platform for building applications that work in conjunction with application servers, databases, and other backend resources to deliver a rich user experience on the desktop.

One of the key benefits of the Eclipse Platform is realized by its use as an integration point. Building a tool or application on top of Eclipse Platform enables the tool or application to integrate with other tools and applications also written using the Eclipse Platform. The Eclipse Platform is turned into a Java IDE by adding Java development components (e.g. the JDT) and it is turned into a C/C++ IDE by adding C/C++ development components (e.g. the CDT [http://eclipse.org/cdt]). It becomes both a Java and C/C++ development environment by adding both sets of components. Eclipse Platform integrates the individual tools into a single product providing a rich and consistent experience for its users.

Integration extends into the rich client space as well. An organization can split up the development of application components across development teams and then integrate the results using the Eclipse Rich Client Platform. This does not trivialize the process of developing large scale applications, but it does make the integration easier. Perhaps the most obvious feature that the Eclipse Platform provides is a managed windowing system. User interface components are part of this (including entry fields, push buttons, tables, and tree views), but there is more. The platform provides window lifecycle management, docking views and editors, the ability to contribute menu items and tool bars, and drag and drop.

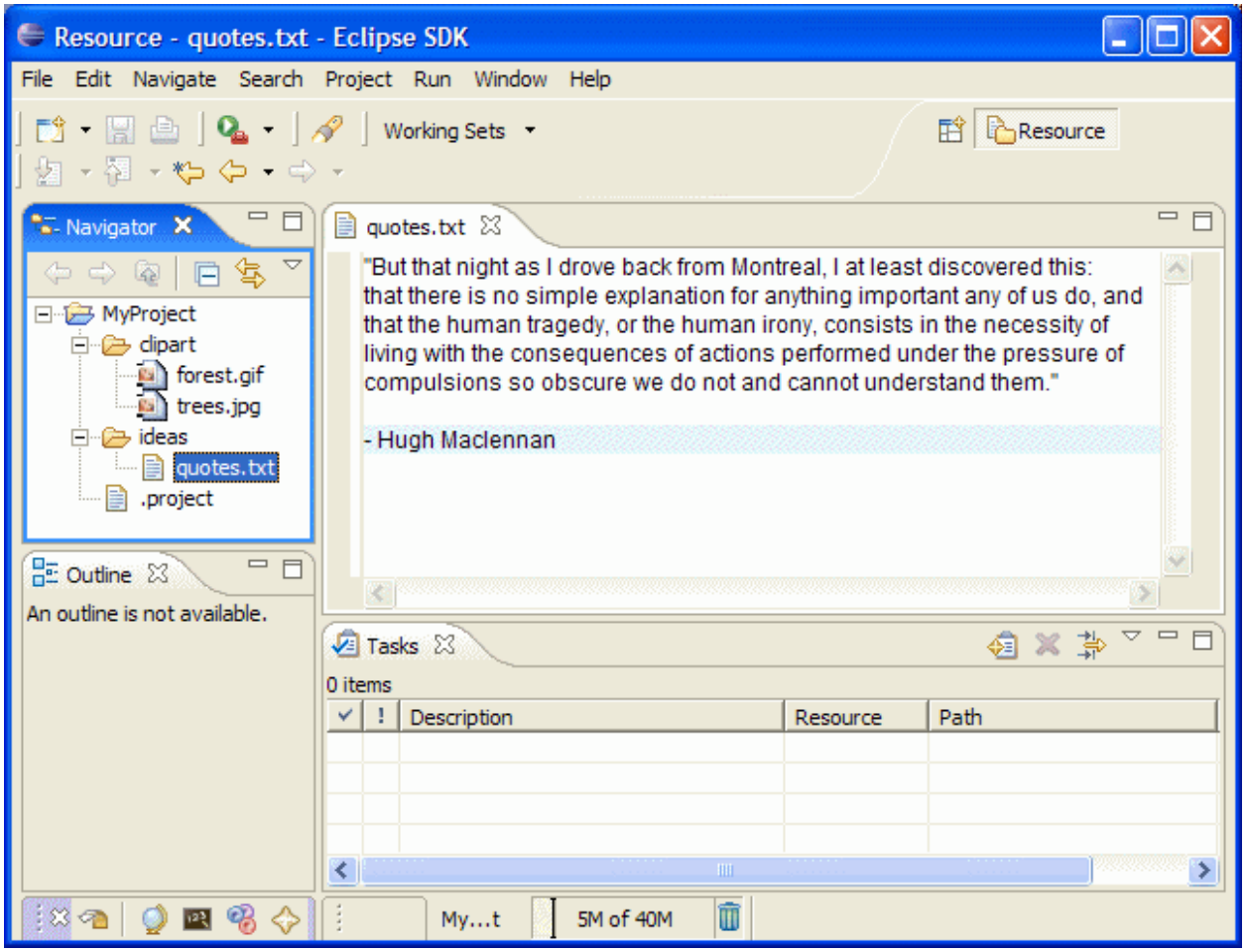

Figure 1-2 the Eclipse Platform User Interface

Figure 1-2 shows a screen capture of the main workbench window as it looks with only the standard generic components that are part of the Eclipse Platform.

The navigator view (Figure 1-2 , top left) shows the files in the user's workspace. The text editor (top right) shows the content of a file. The tasks view (bottom right) shows a list of to-dos. The outline view (bottom left) shows a content outline of the file being edited (not available for plain text files).

Although the Eclipse Platform has a lot of built-in functionality, most of that functionality is very generic. It takes additional tools to extend the Platform to work with new content types, to do new things with existing content types, and to focus the generic functionality on something specific.

The Eclipse Platform is built on a mechanism for discovering, integrating, and running modules called plug-ins, which are in turn represented as bundles based on the OSGi specification. A tool provider writes a tool as a separate plug-in that operates on files in the workspace and surfaces its tool-specific UI in the workbench. When the Platform is launched, the user is presented with an integrated development environment (IDE) composed of the set of available plug-ins. The quality of the user experience depends significantly on how well the tools integrate with the Platform and how well the various tools work with each other.

This plug-ins system allows us to create our own add-in to Eclipse so we can add the functionalities we want. That is what we will do in our project. We will create an Eclipse plug-in to draw Activity Diagrams and transform it into Java Code we can run.

1.4. Project Planning/Control

As we progress in our career, we will face more and more complex and difficult challenges. Some of these may be huge – they may involve the coordination of many different people, the completion of many tasks in a precise sequence, and the expenditure of a great deal of time and money.

The key to a successful project is in the planning. Creating a project plan is the first thing you should do when undertaking any kind of project. Often, project planning is ignored in favor of getting on with the work. However, many people fail to realize the value of a project plan in saving time, money and many problems.

Project Management[7] is the discipline of planning, organizing, and managing resources to bring about the successful completion of specific project goals and objectives. A project is a finite endeavor—having specific start and completion dates—undertaken to create a unique product or service which add beneficial change or added value. This finite characteristic of projects stands in sharp contrast to processes, or operations, which are permanent or semi-permanent functional work to repetitively produce the same product or service. In practice, the management of these two systems is often found to be quite different, and as such requires the development of distinct technical skills and the adoption of separate management philosophy, which is the subject of this article.

The primary challenge of project management is to achieve all of the project goals and objectives while adhering to classic project constraints—usually scope, quality, time and budget. The secondary—and more ambitious—challenge is to optimize the allocation and integration of inputs necessary to meet pre-defined objectives. A project is a carefully defined set of activities that use resources (money, people, materials, energy, space, provisions, communication, motivation, etc.) to achieve the project goals and objectives.

1.5. Code Generation

There are different definitions of code generation depending of what the code is generated from. Our interest is code generated from Activity Diagram so we define our code generation as generation from model. We achieve this with a model transformation that changes our model to a text. This text is the source code of our system.

Dave Thomas of The Pragmatic Programmers, LLC[8] is a well known author and engineer. He says this about code generation:

Just about all of computing is about finding and exploiting leverage. The computer as a tool lets us amplify our abilities in certain, very restricted domains. As programmers, our job is to expand those domains, letting our users leverage computers in new ways.

But we often forget that this leverage is available to us in our own jobs. Sometimes, that is just because we are so familiar with an idea or way of doing things that it never occurs to us

to expand on it. In this way we are like the cobbler's children, generating solutions for other people but not for ourselves.

The leverage of code generators is incredibly important if you are to engineer accurate and maintainable systems. Code generators help in a number of important ways:

1. They help eliminate duplication, and by doing so reduce coupling. With a code generator, one can define an idea or a concept once, and then have that information expressed in numerous different ways throughout the system. Engineer it correctly and this is a prime example of our DRY principle (Do not Repeat Yourself): every piece of knowledge should have a single, authoritative, and unambiguous representation in a system

2. Code generators let us code closer to the user's domain. We can add abstractions using code generators that would be hard to express in just pure Java code. This is important: good code is all about building levels of abstraction and the higher the final level, the easier it is to express the user's intent.

3. Code generators reduce work. Computers do repetitive stuff faster, cheaper, and more accurately than we can. Leibnitz said "it is unworthy of excellent men to lose hours like slaves in the labor of calculation." Let us start using the computers to help us make our jobs easier.

2. Background

2.1. Theoretical Introduction

2.1.1. UML

The Unified Modelling Language became an Object Management Group (OMG)[3] technology in 1997. It was a combination of different well known modelling approaches. In the beginning of the year 2006 the (official) documentation on UML 2.1 was published from the OMG[3]. Within this documentation the new, updated definitions and descriptions of the Activity Diagram was published. The definition was updated in the beginning of April 2006 with the last changes. These changes mainly consisted of editing the current definition text for clarifying its content and making it easier to understand.

Activity Diagrams are diagrams that emphasize the sequence and conditions for coordinating lower level behaviours. But they are also used to describe high level activities. An activity is composed of Actions and each Action can be an Activity. For example, we have an Action that is “sending an invoice”. We could describe this Action further thus make it an Activity that would show details. “Send invoice” Action becomes an Activity where we first get the information, write the invoice then send it. The principle is that you should describe each activity on the high level. You should decompose the activities. To do so you create a new activity diagram to describe this higher level activity with some lower level diagrams and so on until you get a very precise description of the system. These higher level activity diagrams are commonly called control flow and object flow models. The actions in an Activity Diagram are initiated when other actions finish, when a new events occurs external to the flow or data and objects become available.
Some find activity diagrams very close to Petri nets[9] or state diagram[10] but they are somewhat different. When we reach a state in a state diagram, we are waiting for a trigger to move forward but the activities of an activity diagram are not waiting. When the Action is finished, then we go to the next.

As for comparison to Petri nets, Rik Eshuis[11] says that the main difference is that activity diagrams are not intended for workflow modelling because the semantics are not formal. His goal is to use these semantics for analysing workflow models in activity diagram notation by means of model checking.

The different parts of an Activity Diagram are called activities and actions. It means that an activity may contain other activities. In that case they should be drawn in their own activity diagram. An activity diagram has different levels with different metaclasses: fundamental, basic, intermediate, complete, structured, complete structured and extra structure.
The fundamental level defines activities as nodes which include actions. The basic level includes control sequencing and data flow between actions but explicit forks, joins, decision and merge are not supported. The intermediate level adds the support for the fork, join, decision and merge to the basic level and it requires the basic level. The complete level includes constructs that enhance the lower level models. The structured level supports modelling of traditional structured programming constructs, such as sequences, loops, and conditionals, as an addition to fundamental activity nodes. It requires the fundamental level

and is compatible with the intermediate and complete levels. The complete structured level adds support for output pins of sequence, conditionals and loops. It also depends on the basic level for flows. And the extra structure level supports error handling and requires the structure level.

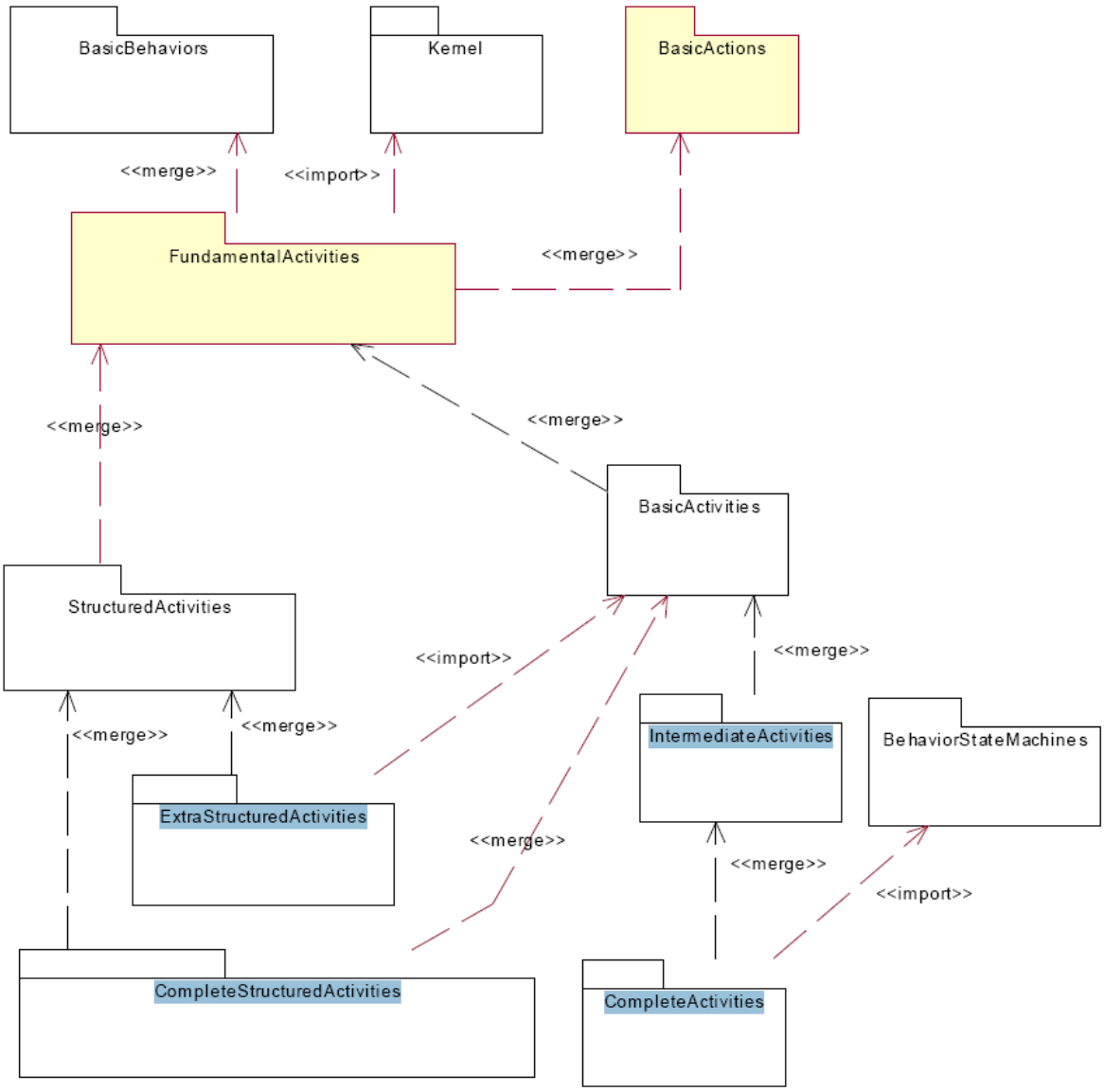

Figure 2-1 Dependencies of the Activity packages

The Figure 2-1 shows the dependencies of the activity packages. This is only the example for a complete structured level. There are 23 other metamodels describing the activity packages in the different levels.

You can find metamodels for every part of the Activity Diagram but we will not show all of them here.

To fully understand the UML Activity Diagram, we should explain every class that it contains. There are 52 classes so we shall present only some of them here, not all. See the UML documentation for complete reference[3].

2.1.1.1. Action

An action represents a single step within an activity which means that an action is not further decomposed. However, a call behaviour action may reference an activity definition in which case the execution of the action involves the execution of the activity with all its actions.
It may have sets of incoming and outgoing activity edges that specify control flow and data flow from and to other nodes. An action will not begin execution until all of its input conditions are satisfied. The completion of the execution of an action may enable the execution of a set of successor nodes and actions that take their inputs from the outputs of the action. It may be extended to have pre- and post conditions. These are constraints that should hold when the execution of the action starts and completes.

In an Activity Diagram, an action looks like a round-cornered rectangle. The pre- and post conditions are viewed as notes with keywords «localPrecondition» and «localPostcondition»

2.1.1.2. Activity

An activity is the specification of parameterized behaviour as the coordinated sequencing of subordinate units whose individual elements are actions and/or activities. It specifies the coordination of executions of subordinate behaviours, using a control and data flow model. The subordinate behaviours coordinated by these models may be initiated because other behaviours in the model finish executing, because objects and data become available, or because events occur external to the flow. Activities can contain actions of different types. Most of the constructs in the Activity Diagram deal with various mechanisms for sequencing the flow of control and data among the actions.

The notation for an activity is a combination of the notations of the nodes and edges it contains, plus a border and name displayed in the upper left corner.

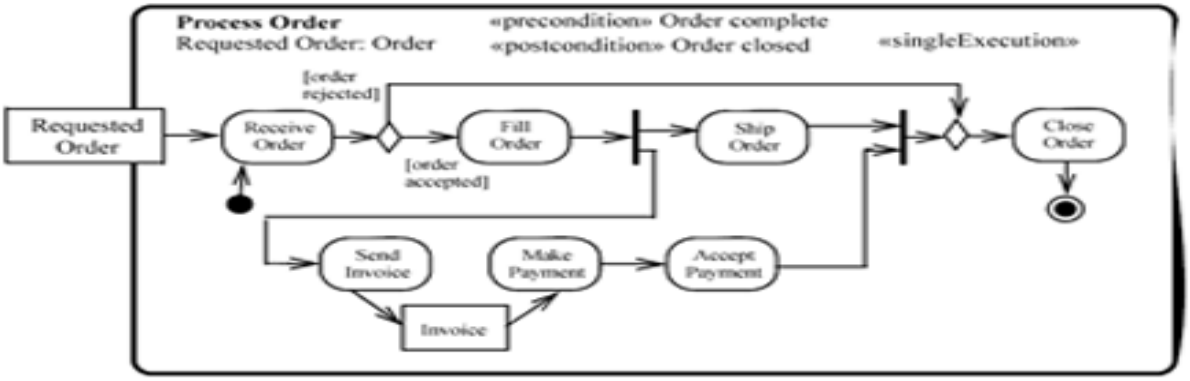

Figure 2-2 an example of an activity diagram

Figure 2-2 shows how an activity diagram may look like with the most common elements.

2.2. Eclipse

Eclipse[5] was chosen for two main reasons. First, it is free and a well known platform recognised by many as a high standard for development and providing tools for development on the platform, The Eclipse Plug-in Development Environment (PDE). Second, it provides with powerful tools for modelling: the Eclipse Modelling Framework (EMF).

2.2.1. Eclipse Modeling Framework (EMF)

EMF[1, 12] is a modeling framework and code generation facility for building tools and other applications based on a structured data model. It provides tools to produce a set of Java classes from a model and a basic editor. It consists of three parts: the core framework, the edit framework and the code generation facility. In EMF the models used are called ecore models (see Figure 2-3 for an example).

The core framework includes a metamodel for describing models and runtime support for the models, the edit framework includes classes for building editors and the code generation

facility is capable of generating everything needed to build a complete editor for an EMF model. The framework includes a GUI from which generation options can be specified.

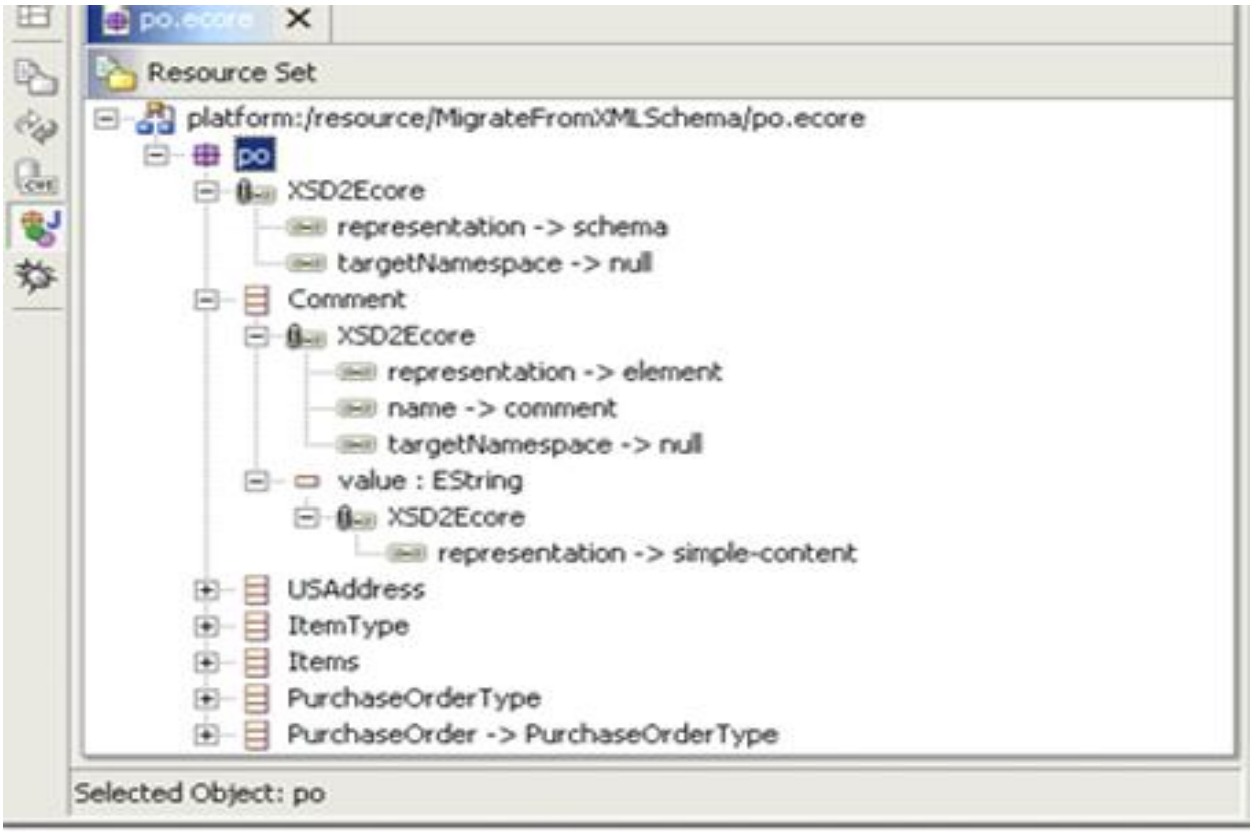

Figure 2-3 an ecore model

The ecore model here (Figure 2-3) is shown as a tree. That is one of the many ways you can see it in Eclipse.

EMF consists of three fundamental pieces:

- **EMF** - The core EMF framework includes a meta model (Ecore) for describing models and runtime support for the models including change notification, persistence support with default XMI serialization, and a very efficient reflective API for manipulating EMF objects generically.
- **EMF.Edit -** The EMF.Edit framework includes generic reusable classes for building editors for EMF models. It provides:

- Content and label provider classes, property source support, and other convenience classes that allow EMF models to be displayed using standard desktop (JFace) viewers and property sheets.
- A command framework, including a set of generic command implementation classes for building editors that support fully automatic undo and redo.

- **EMF.Codegen** - The EMF code generation facility is capable of generating everything needed to build a complete editor for an EMF model. It includes a GUI from which generation options can be specified, and generators can be invoked. The generation facility leverages the JDT (Java Development Tooling) component of Eclipse.

Three levels of code generation are supported in EMF:

- **Model** - provides Java interfaces and implementation classes for all the classes in the model, plus a factory and package (meta-data) implementation class.
- **Adapters** - generates implementation classes (called ItemProviders) that adapt the model classes for editing and display.
- **Editor** - produces a properly structured editor that conforms to the recommended style for Eclipse EMF model editors and serves as a starting point from which to start customizing.

All generators support regeneration of code while preserving user modifications. The generators can be invoked either through the GUI or headless from a command line.

2.2.2. The UML2 Plug-in

The UML2 plug-in[13, 14] is an EMF-based implementation of the UML 2 metamodel. It provides a Java API support to manipulate UML 2.0 models. The UML2 provides an implementation to support the development of modelling tools with a common XMI schema and validation rules to define levels of compliance. It has been updated the summer 2006 to UML 2.1, so we worked with an updated model in our project.

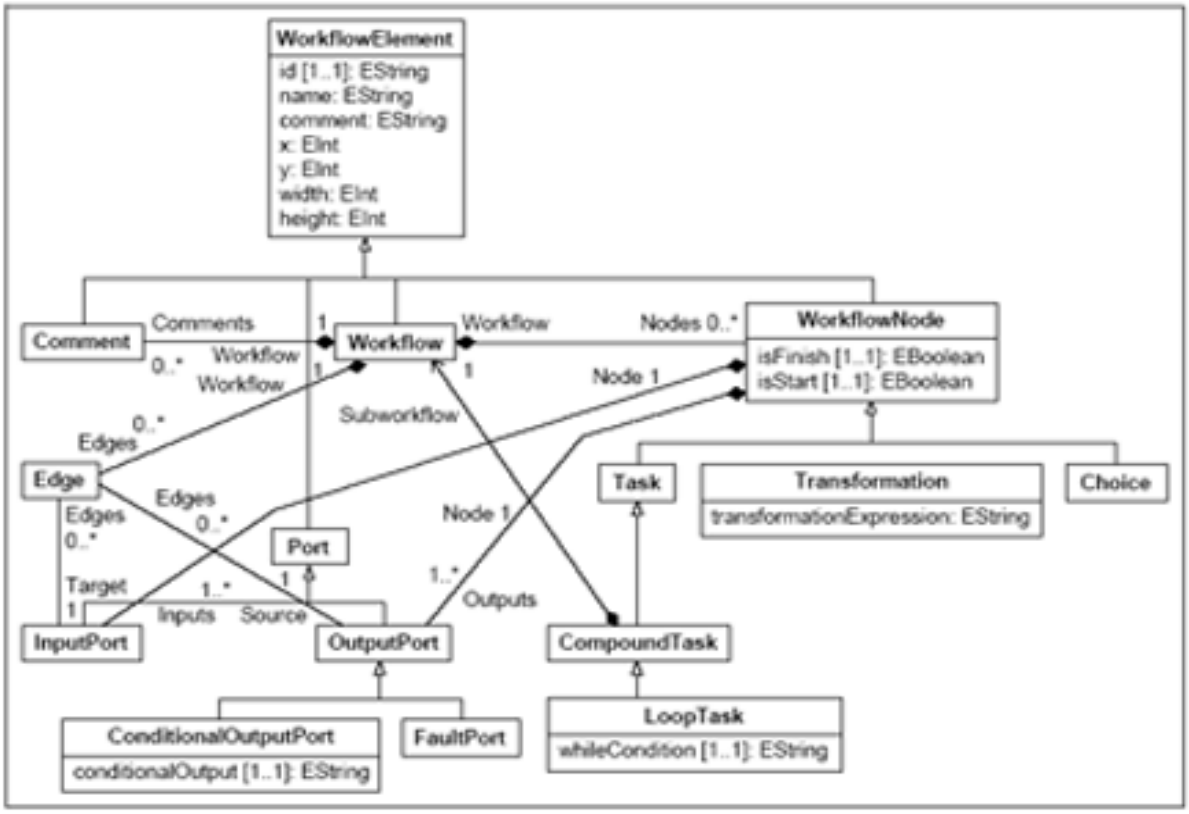

Figure 2-4 A UML model in Eclipse

Figure 2-4 shows how the different elements are represented in the Eclipse UML plug-in.

The Eclipse project also includes a set of GMF-based editors for viewing and editing UML models called UML2 Tools

2.2.3. Eclipse Plug-in Development Environment (PDE)

When creating a plug-in for eclipse, you have to create your plug-in manifest file (plugin.xml), specify your plug-in runtime and other required plug-ins, define extension points, include their specific mark-up, associate XML Schema files, etc. The PDE project provides a number of views and editors that make it easier to build plug-ins for Eclipse. It is divided in two parts: the build and the user interface (UI). The build facilitate the automation of plug-in build processes and the UI provides wizards for creation, imports, exports, build and deploy, an editor, a launcher to test and debug, a search engine and a log.

One of the many wizards in PDE:

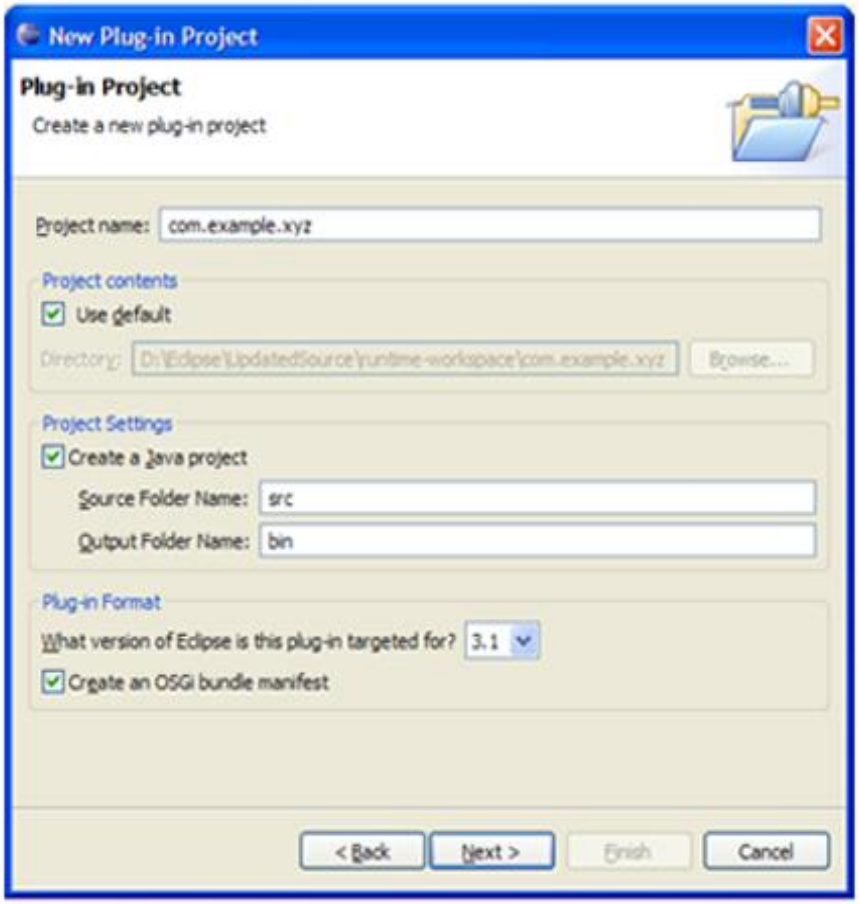

Figure 2-5 the new plug-in project wizard in Eclipse

Figure 2-5 shows the first window from the wizard creating new plug-in projects.

2.2.4. GMF

The Eclipse Graphical Modeling Framework (GMF) [2] provides a generative component and runtime infrastructure for developing graphical editors based on EMF and GEF. For example, a UML modeling tool, workflow editor, etc. Basically, a graphical editing surface for any domain model in EMF you would like. The project aims to provide these components, in addition to exemplary tools for select domain models which illustrate its capabilities. The Graphical Modeling Framework (GMF) aims to allow developers to create a rich graphical modeling-oriented editor from an existing application model.

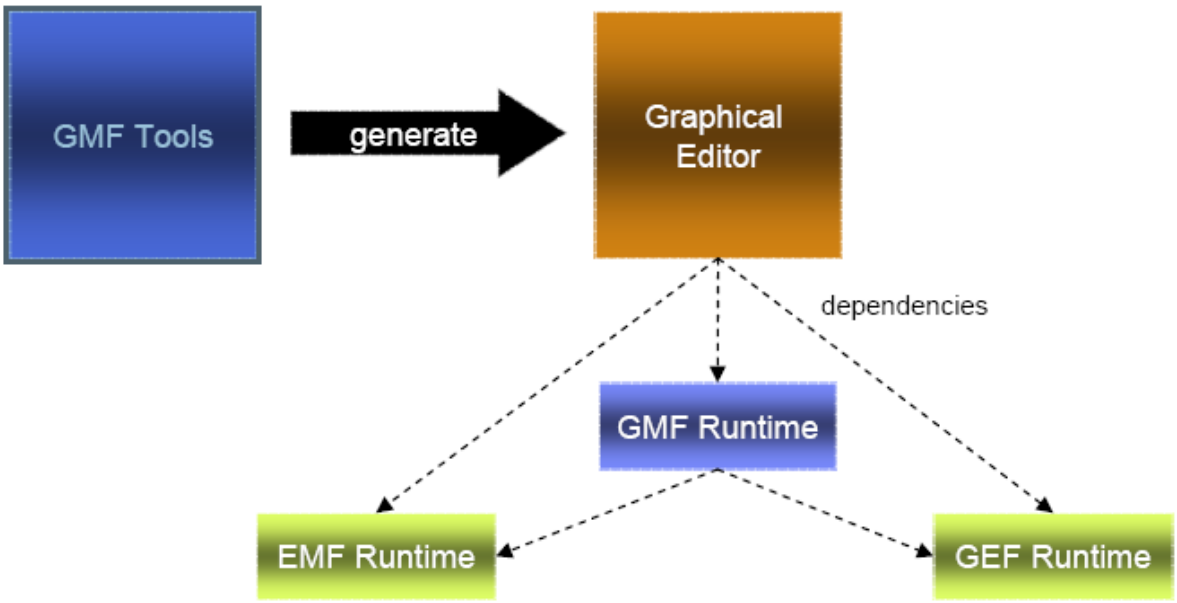

Figure 2-6 Shows how GMF is running

It is one of sub-projects in top-level Eclipse Modeling Project. This project was announced at EclipseCon 2005. First version of GMF was released as a part of the Callisto Simultaneous Release in June 2006 with an upcoming maintenance release at the end of September. The second major release was 2.0 and was a part of Europa Simultaneous Release. As it is specified, GMF consists of two parts – generative and runtime. The runtime part could be described as a set of plug-ins extending existing EMF and GEF functionality. The runtime allows easier integration between EMF and GEF, and provides additional services like: transactions support, extended meta-modeling facilities, notation meta-model, variability

points used for runtime extensibility of generated code, etc. The GMF Runtime requires EMF and GEF as it binds the two to make them easier to use.

The generative part mainly provides users with the possibility to define future diagrams using specially designed EMF models and generate code using this information.

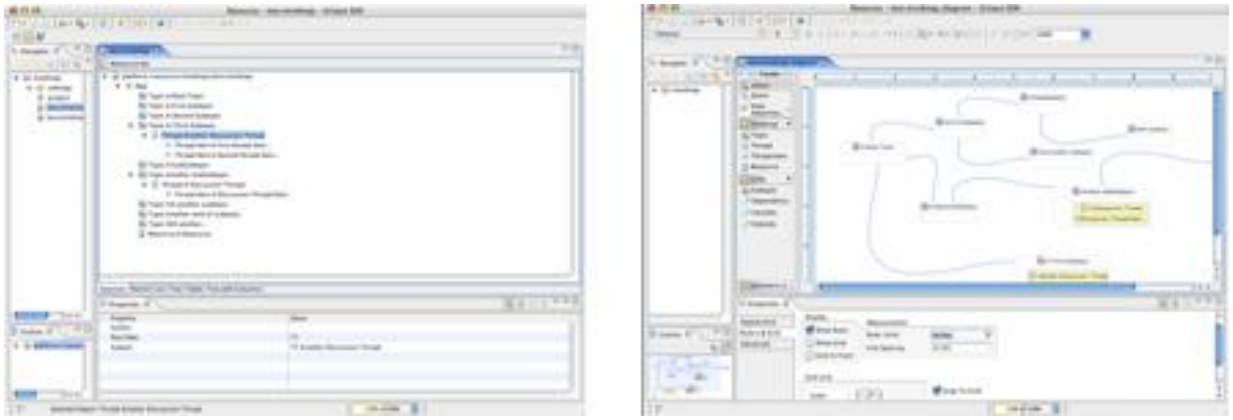

Figure 2-7 Difference between before and after GMF

GMF also offers many reusable components for graphical editors like Action Bars, Connection Handles, Collapsed Compartments, Geometric figures, Toolbar, Property view, Actions and many more. We can also notice that code generation is made with Java Emitter Template (JET) but we used MOFScript for our code generation. GMF requires minimal coding from the user compared to what it was with GEF. The users only need to design some models to define what they want to do. There are six models in GMF all are EMF models saved as XML and GMF user takes advantages of the EMF editor to edit them:

- The center of any GMF project is the Domain Model where the user specifies the meta-model he or she will work with. From there, the user can use wizards to help design the remaining models or design them with the EMF editor.
- The Domain Gen Model is directly derived from the domain Model and is used in code generation.
- The Graphical Def Model is where the user will define how every Node, Link and Label will be represented.
- The Tooling Def model is where the user will define which element will be available in the side bar of the editor. This is called Palette in Eclipse.

- The Mapping Model where the user will map elements together. A simple example is to take an element from the meta-model that he or she has in the Domain Model. Map it to a graphical representation from the Graphical Def Model and add a tool from the palette to it defined in the Tooling Def Model.
- The last model is the Diagram Editor Gen Model. It is transformed from the Mapping Model and is the model the user will transform plug-in code from.

For a complete overview of the models used in MADE and how to move forward through GMF, see the Appendix.

2.2.5. MOFScript

MOFScript[15] aims at developing tools and frameworks for supporting model to text transformations, e.g., to support generation of implementation code or documentation from models. It should provide a metamodel-agnostic framework that allows usage of any kind of metamodel and its instances for text generation. MOFScript is an Eclipse-based text transformation tool and engine. It is developed as part of the EU projects MODELWARE and MODELPLEX. It comes with an Eclipse lexical editor and a transformation engine for generating text.

The MOFScript language is currently a candidate in the OMG RFP process on MOF Model to Text Transformation. The MOFScript subproject aims to support the MOFScript language in terms of editing, parsing, and execution, and will aim to comply with the direction of the possible future technologies.

The subproject will cover the aspects needed in the context of text generation in software engineering, including:

- Generation of text from MOF-based models: The ability to generate text from any MOF-based model, e.g., UML models or any kind of domain model.
- Control mechanisms: The ability to specify basic control mechanism such as loops and conditional statements.
- String manipulation: The ability to manipulate string values.

- Output of expressions referencing model elements.
- Production of output resources (files): The ability to specify the target file for text generation.
- Traceability between models and generated text: The ability to generate and utilize traceability between source models and generated text, e.g., to provide re-generation.
- Ease of use: An easy-to-use interface.

Reverse engineering is kept out of scope at this time, but may be included in the scope at a later point in time.

At this time, code has been developed that supports parsing, checking, editing, and execution of MOFScript code. Currently, these are deployed as separate Eclipse plug-in components, which run within Eclipse. The parser and run-time may also be used independently of Eclipse.

The following code is ready for submission to an Eclipse MOFScript subproject:

- MOFScript model module: The MOFScript model is an EMF/ecore model, which is the target of the parser and the subject of the runtime/execution engine.
- Parser module: The parser reads MOFScript text and produces an instance of the MOFScript model, which is also semantically checked.
- Runtime module: The runtime provides an execution environment for a MOFScript model. It handles all aspects of execution.
- Editor module: The editor provides an Eclipse editor environment for the MOFScript language, including content assist, preferences, outliner, documentation and more.

An initial user guide and examples will also be made available.

The MOFScript tool is developed as two main logical architectural parts: tool components and service components. The tool components are end user tools that provide the editing capabilities and interaction with the services. The services provide capabilities for parsing, checking, and executing the transformation language. The language is represented by a model (the MOFScript model), an EMF model populated by the parser. This model is the basis for

semantic checking and execution. The MOFScript tool is implemented as an Eclipse plug-in using the EMF plug-in for handling of models and metamodels.

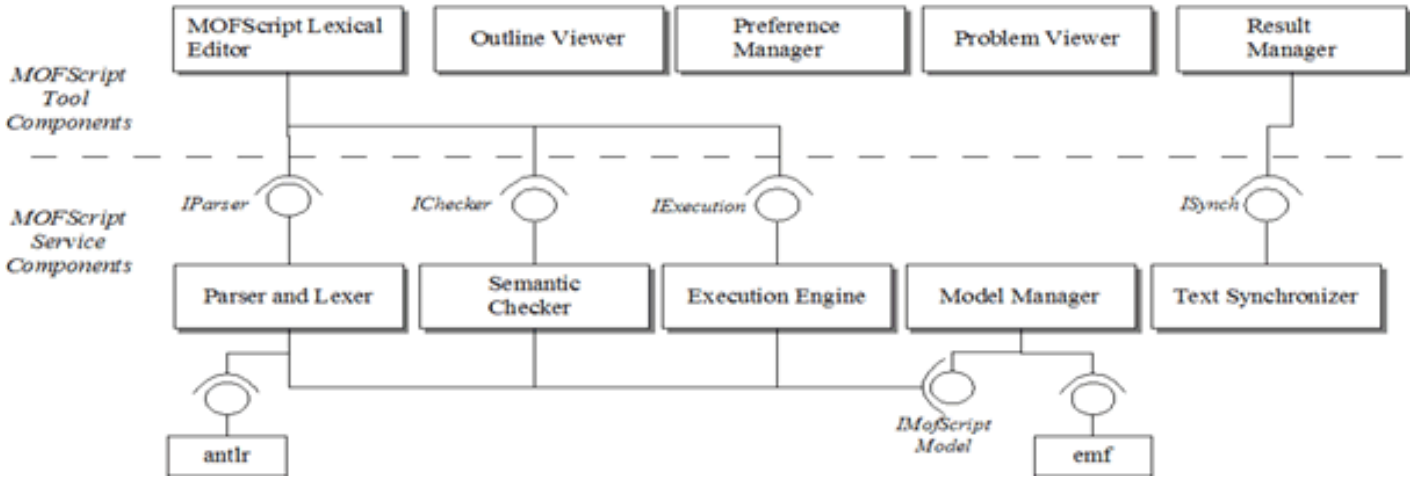

Figure 2-8 MOFScript plug-in architecture

The Service Components consist of these component parts: The Model Manager is an EMF-based component which handles management of MOFScript models. The Parser and Lexer are responsible for parsing textual definitions of MOFScript transformations, and populating a MOFScript model using the Model Manager. The parser is based on antlr. The Semantic Checker provides functionality for checking a transformation's correctness with respect to validity of the rules called, references to metamodel elements, etc. The Execution Engine handles the execution of a transformation. It interprets a model and produces an output text, typically to a set of output files. The Text Synchronizer handles the traceability between generated text and the original model, aiming to be able to synchronize the text in response to model changes and vice versa.

2.3. Related Tools

We will see that there are many actors trying to use UML for different purpose. Most tools we find are tools that allow a user to draw a diagram. There are few that try to generate code from a diagram and even less that use Activity Diagram. Here are some examples of UML tools. For this part of my research we needed to find relevant tools to compare with. We

selected these because they were mentioned in other research we made for this work and were ranked very high in the search engines we used. And they are also based on Eclipse. We removed others from this list because they were not accessible to me, since there were no trials or free versions available.

2.3.1. MagicDraw

MagicDraw is a visual UML modeling and CASE tool with teamwork support. It provides code engineering mechanism (with full round -trip support for J2EE, C#, C++, CORBA IDL programming languages, .NET, XML Schema, WSDL), as well as database schema modeling, DDL generation and reverse engineering facilities. It supports many technologies like UML, SysML 1.0, OCL, Java 2, CORBA IDL, EJB 2.0, C#, CIL (MSIL), C++, JDBC, XMI 2.1, EMF UML2 2.x XMI. Furthermore, it also provides code generation and reverse engineering.

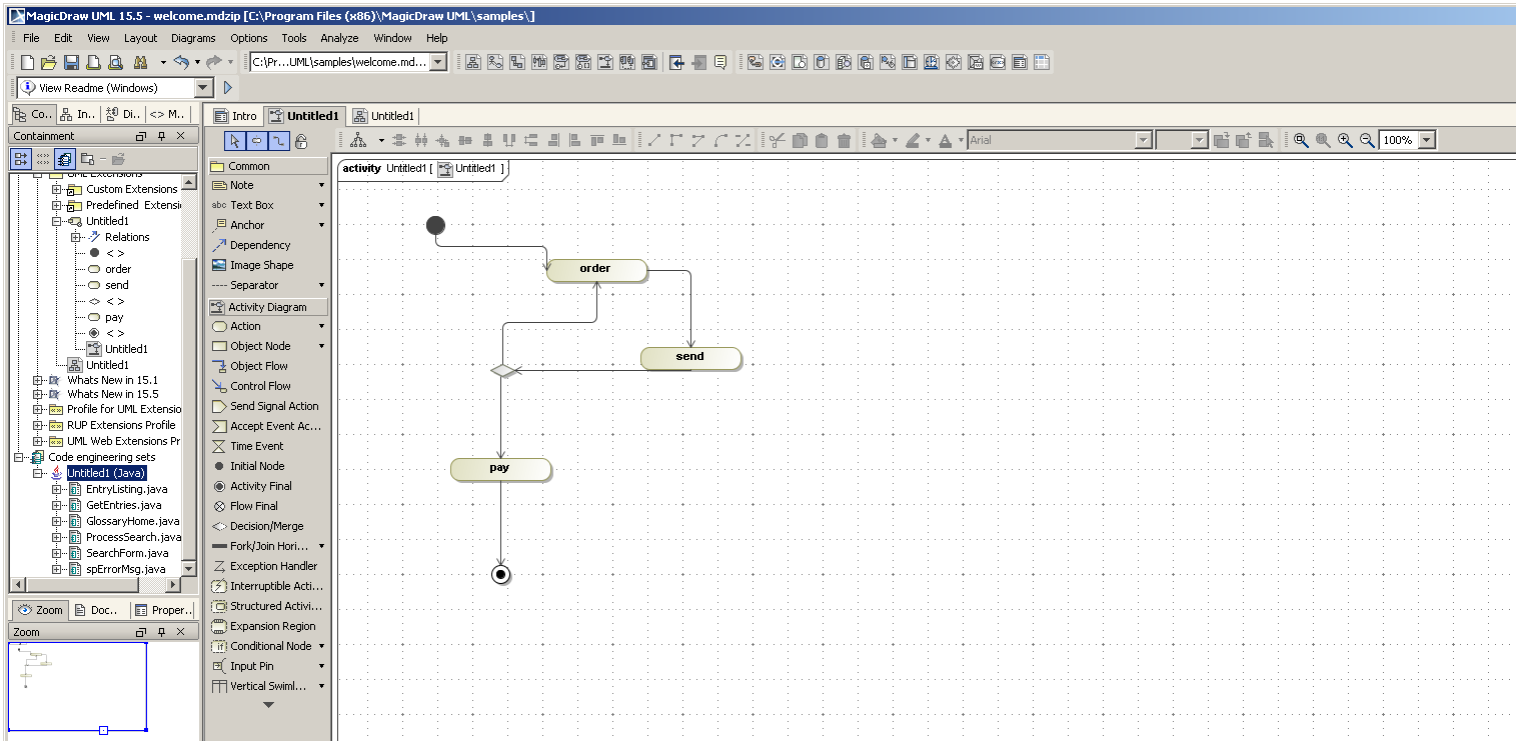

Figure 2-9 An Activity Diagram in MagicDraw

MagicDraw is a huge software package with much functionality. It is not easy to understand how we should move forward. As many other proprietary products, there are many protective

measures, such as key to download, many required license agreements etc… All these restrictions give the product an inconvenient and complex impression. We used the demo version and we could not even save our changes without pop-ups to inform us about the restrictions and how we could buy. After that, we tried to create the Activity Diagram and it went quite nice. Then we tried to generate code, but there were still restrictions due to the demo license. We cannot afford to buy the Professional license needed to run code generation. As a result, we could not try it ourselves. After some more research, it seems it is only Class Diagrams that is supported for code generation. From what we could see, it is a fully scaled system that can do a lot, but this makes it difficult to use.

2.3.2. Rational Software Modeler

Rational Software Modeler enables users to specify and communicate development project information from several perspectives and to various stakeholders. It is based on Eclipse 3.3 and provides support for modeling with the UML 2.1 as well as for creation of UML-based Domain Specific Language environments and "simplified" UML modeling environments. In addition, there are other interesting features like:

- Helping ease the transition between the architecture and the code with model-to-model and model-to-code transformations.
- Including reverse transformations.
- Allowing you to apply included design patterns.
- Authoring your own design patterns to ensure that conventions and best practices are followed.

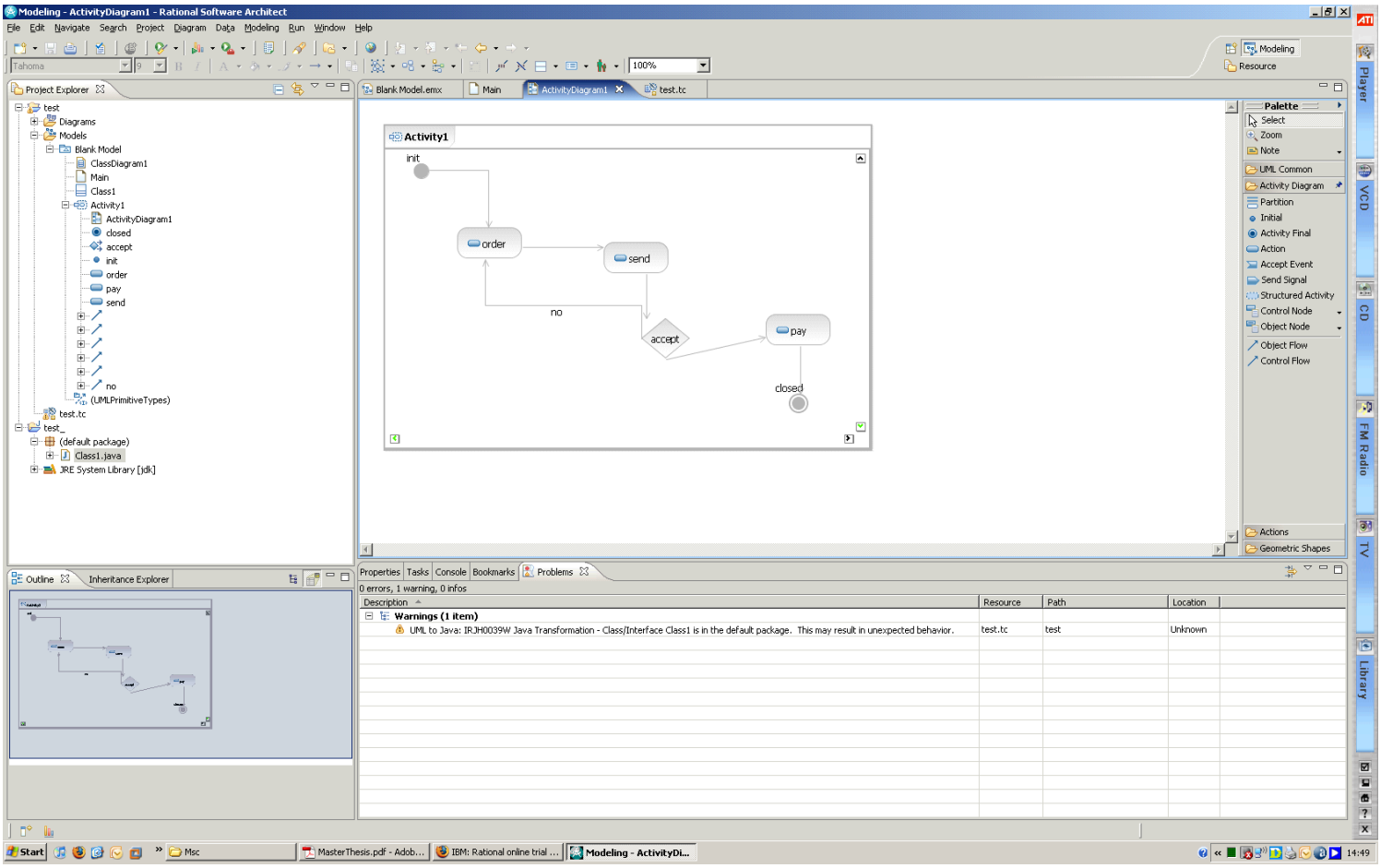

Figure 2-10 An Activity Diagram in Rational Software

When we tried Rational Software, it is easy to start and we created an Activity Diagram within minutes. Then we created a transformation and ran it. Nothing happened. We tried different possibilities and found out that the standard transformation that Rational Software offers is only for models, packages, Classes and Interfaces. But we can, of course, create our own transformation to achieve transformations from Activity Diagram but we do not have any control over the model created when we add new diagrams and Rational seems to add every Diagram to the same model. For us, this means that we would have to parse all diagrams added to our model and not only the Activity Diagram. This is not a problem but it would create a bigger load when transforming because there are more items to check.

2.3.3. Borland Together

Borland has a goal to provide Visual Modeling for Software Architecture Design with this product. It provides modeling with domain-specific languages that you can create and deploy using Together's Domain Specific Language (DSL) toolkit. It also allows creation of UML 2 and business process models (BPMN) to generate and import business process execution languages with Web Services definitions (BPEL4WS). This program also offers code generation and code reviews.

It also has support for OMG's Query View Transformation (QVT) used in model-to-model transformations and support for OCL 2.0 with syntax highlighting, validation, code sense, debugging and expression evaluation. We had some problems with this software. The installation was full of bugs and it took a long time before we could start the application.

2.3.4. MyEclipse UML

MyEclipse UML is part of the MyEclipse[16] package. It is now version 6 and contains Use-case, class, sequence, collaboration, state, activity and Deployment diagram editor. It also includes a free-form drawing tool which generates Java code. It is much more than a UML tool but, to get the UML functionality, we need the professional version and it costs 59.95$ per year. It includes many features like JavaScript editor and debugger, Ajax Tool, some database support, visual web designer, Spring, XML, web services, Struts designer, Java Server Faces (JFS) and many more features.

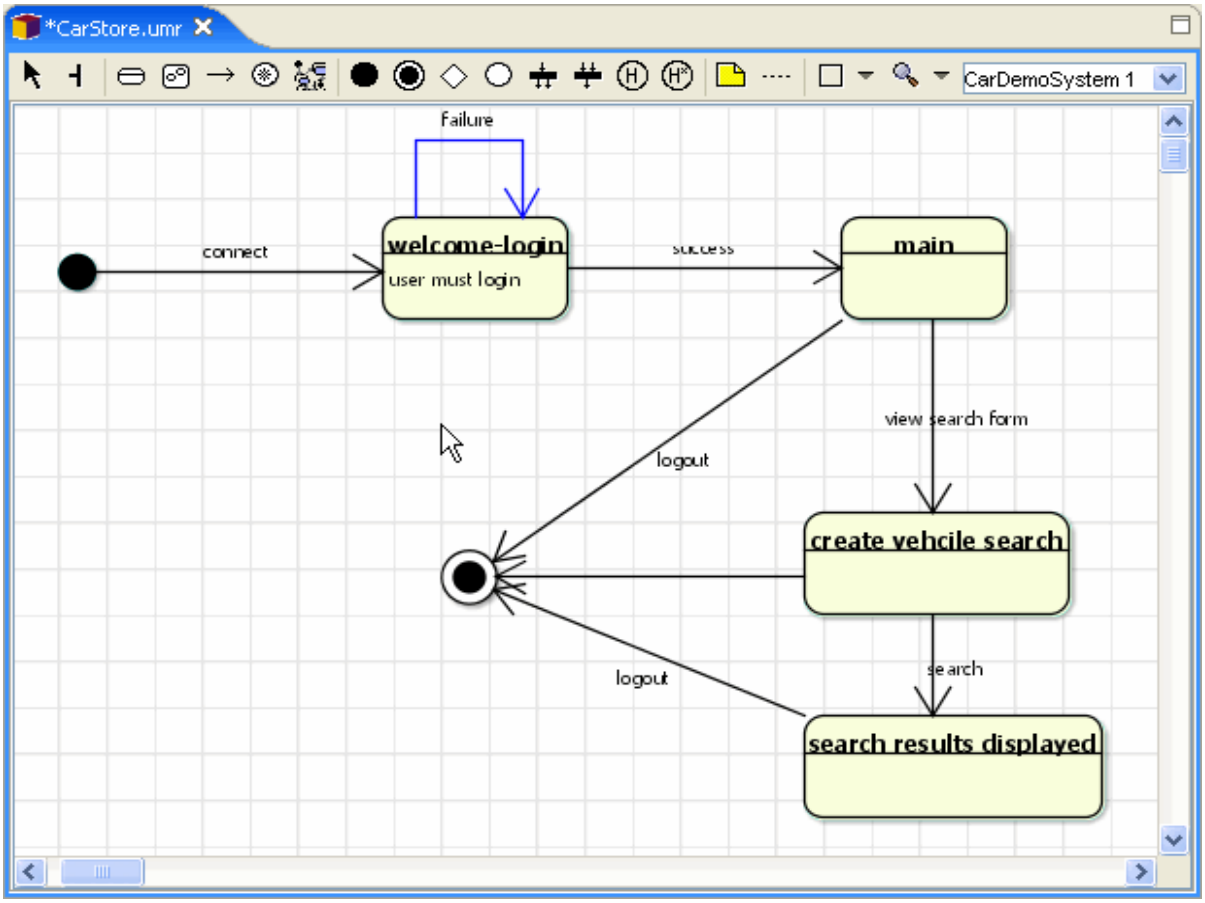

Figure 2-11 A MyEclipse diagram

In this example (Figure 2-11), you can see that it is a quite simple and clear design. It generates Java code but only from Class Diagram Not from Activity Diagram. This is a

commercial Eclipse package so you can use much functionality but you also get many plug-ins that you probably will never need.

This product allows the user to do many different things. For example, you can write Java code and web development, in addition to UML. But it becomes a very big package if you want to use it only for UML. We would prefer a single plug-in. And it does not generate code from anything else than Class Diagrams so we will be adding a new functionality to Eclipse with our plug-in compared to MyEclipse.

2.3.5. Poseidon for UML

There are four editions available: the community, the standard, the professional and the embedded edition. None are updated for UML 2.1. They are still working with 2.0. In the current version, 6.0.2, you can generate code with the community edition from Class diagram only and they removed most of the existing functionalities for Java in Eclipse. All versions are stripped down to be a UML tool and nothing more. You will have to buy the standard edition to be able to add plug-ins. And the system requirements say that it is only from their technology partners. We would really like to still be able to customize Eclipse as we like. It is, in my opinion, one of the main advantages of this platform. It has some nice functions to help creating the diagram like hints when clicking on an element. Furthermore, it also has a quite easy and clean layout for design but it is quite hard to understand the element tree (on the left in Figure 2-12). It is clear at the top level where it only lists the different types of diagrams you have. But when you start expending the tree, you will see that it lists everything. Even the font you used (if you changed from the standard) or size of elements. It gets a bit confusing when you have a huge diagram. In addition, it also generates code to C++ and C#.

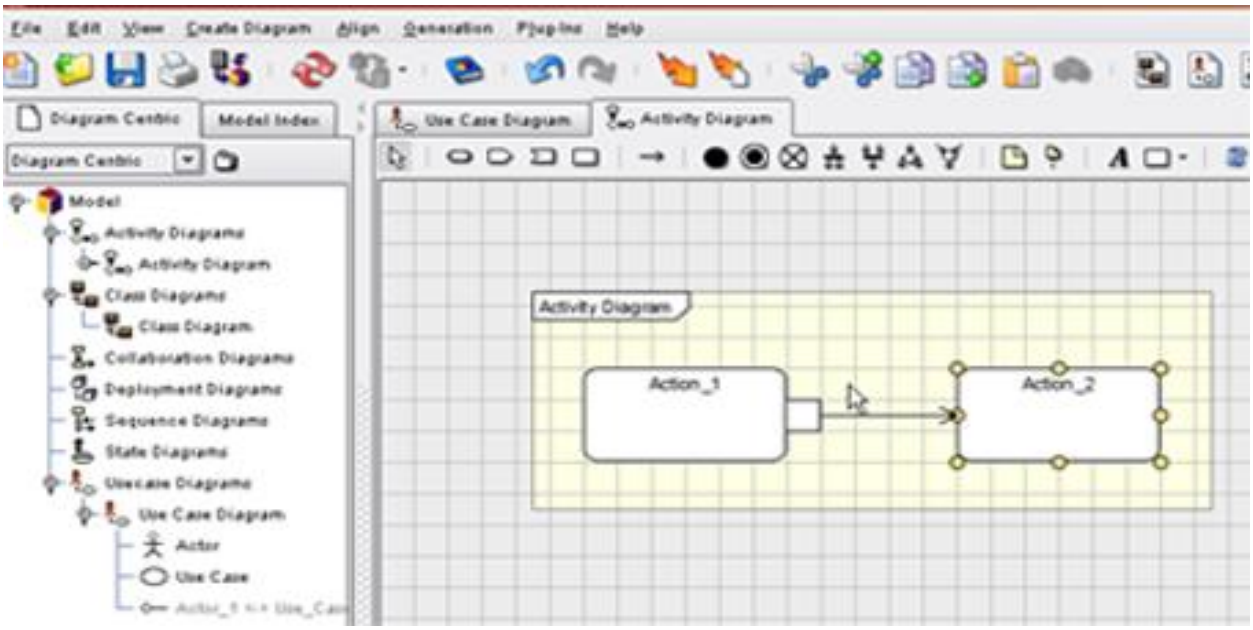

Figure 2-12 Poseidon for UML workbench

Figure 2-12 shows an example of an activity diagram and most of the workbench in Poseidon for UML.

Once again, the product does what it is supposed to do but it is only for UML design and we will have to pay to get something more. Our plug-in could actually be an addition to Poseidon.

2.3.6. PapyrusUML

Papyrus[17] is a tool for Graphical UML Modelling. It supports UML2 and has plug-ins for MARTE, SysML, CCM and LwCCM UML profiles. We downloaded it (it is free open source) and started it. There is a model editor and a graphical editor. We could not find support for anything other than Class Diagram at first. You actually need to create the hierarchy in the Outline view to be able to create other diagrams. PapyrusUML is a very complete and updated UML tool. It looks really complete with many UML features like UML profiles and Diagram Interchange. Even though we could not find any code generation from diagrams, you can create some code generators with the ANTLR plug-in that is included in Papyrus UML. There is support for the whole UML meta-model but when we added something to the Class Diagram in the model view and made changes in the graphical view, the elements added in the model view were gone.

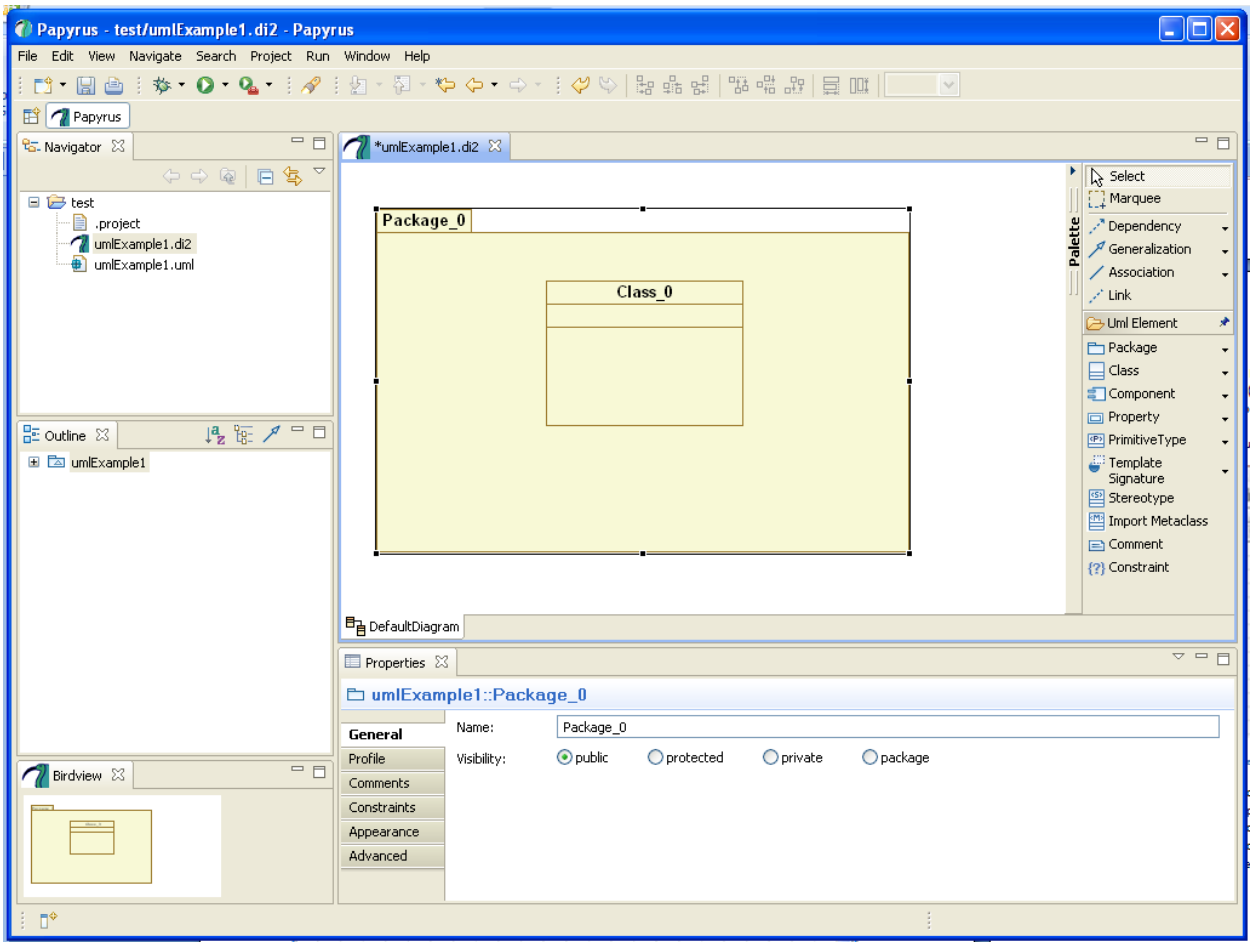

Figure 2-13 Class Diagram in Papyrus UML

I think it is a really good UML tool but you need to create your own code generators and it is a bit difficult to understand how it works. There are wizards for new diagram creation. You need to go in the outline view and right click in the hierarchy to create one. We think that our plug-in would be a good addition to this program. Although we perceive the applications objectives to be different.

3. Definition

3.1. Model

A model is at the first level of abstraction above the items of interest to the modeler. A metamodel is at the second level of abstraction for a modeler and has the modeling constructs themselves as its items of interest. So, a metamodel is a special type of model.

The reason why we need more than one metamodel for all modelers is that there are different types of models. For example, the constructs/rules used to model process flows are different than the constructs/rules used to model data structures. Please note that an explicit metamodel of the constructs/rules used by modelers is not always documented. Frequently, modelers invent their own notations and meanings and use them implicitly in their models. This usually leads to misunderstandings and arguments over what the model means. That is why it is important to use modeling constructs/rules that are documented in an explicit metamodel.

There are always 3 levels of abstraction for any modeling context. First, there is the model itself (level M1), there are instances of the model (level M0), and there are a set of constructs/rules for constructing the model (level M2). It can be confusing when the instances of a model and/or the rules for constructing that model can be viewed as items of interest in another model. If you shift the level of abstraction in order to look at instances as a model or at modeling constructs/rules as a model, then you have reset the M1 level of your modeling context.

Some people talk about meta-metamodels (level M3) as being a third level of abstraction for a modeler. There are contexts (e.g., modeling tools) where models are viewed (e.g., imported/exported) as instance data. Because models are at level M1 in most contexts (but actually at level M0 in the modeling tools context), it is usually less confusing to refer to them as at level M1 regardless of context instead of as at level M0 (or as at level M2!) in the atypical contexts. So, the constructs/rules used to describe the (meta) model B for a modeling tool can be viewed as a meta-metamodel at the third layer of abstraction for a modeler, even though they are actually at level M2 in the modeling tool context.

3.2. Ecore

The ecore is a meta-metamodel used by Eclipse EMF projects. It defines how the models should work with the EMF plug-in. For full details, you can have a look at the Java API[18].

As for our project, we use the UML.ecore. This is the EMF definition of the UML superstructure.

3.3. Resource

As a key concept of our project we should define resource. There are many definitions because its meaning is slightly different in different cases. For example, resource of a computer is often defined in terms of CPU time, memory, hard disk, etc… But we are talking of totally different thing when thinking about "natural resources". It shows that the need for a definition in our case is quite obvious.

There are numerous definitions and we will have to restrict ourselves. We should mention first how the Word Net Encyclopedia defines resource:

Noun

- S: (n) **resource:** available source of wealth; a new or reserve supply that can be drawn upon when needed.
- S: (n) **resource**: a source of aid or support that may be drawn upon when needed. *"The local library is a valuable resource".*
- S: (n) **resource**: resourcefulness, imagination, the ability to deal resourcefully with unusual problems. *"A man of resource".*

Let us analyze this definition. We think it is way too general for us. It does not help to confine our intended meaning. So we will need to adjust it. We think that a good way is to check other definitions and filter what is useful. So here are others.

Wikipedia[19] have 3 different groups of definitions:

1. Computer science: this definition talks about very specific resource as CPU, Memory, Hard disk, etc... [20]

2. Economics: Here, it is defined as Land, Labor and Capital goods (or Natural, Human resources and services)[21]
3. Resource types and development: This one is a very general resource definition[22] that says that a resource is something useful.

If we continue our analysis with number one here, we can see that it would be relevant for us. Imagine that our Activity Diagram will represent a computer system; it is then a very good definition. The problem is that we do not want to narrow our tool to computer system design so we will need to find a more general definition that can apply to a business for example.

When we look at the number 2, we can see that it has a more general definition. Those resources can be used, of course, but we think that it does not define it precisely enough. We will have a look at the third one to emphasize my point.
The number 3 defines a resource as something that fulfills these three conditions:

1. They have utility[1]
2. Many of them are found in limited quantity
3. They can be used to provide services and produce new goods

This is a quite good definition for us because it suits our need. The other ones define resources more economically. They have a specific definition that is not general enough for us. In our mind, the last one is general enough to integrate every type of resources we thought of but still precise. We will use this last one as a starting point and define resource in MADE as such.

3.4. Constraint

We want to base the constraints on the resources. We think it is the best way for a planning tool since it is central in every project and optimization. The best is always to use as few resources as possible.

[1] Utility is defined as the relative satisfaction from consumption of goods. Given this measure, one may speak meaningfully of increasing or decreasing utility, and thereby explain economic behavior in terms of attempts to increase one's utility.

We want to define a constraint as a condition on one or more variables that represent the resource(s). The test can be a simple test as time < 50 days. It can be a very complicated test that implicates variations in many variables. Let us explain it with an example. The constraint would be connected to different actions or at the start and the end of one action. We take "samples" at those connection points. This means that we check how much time or money (depending on the resource we check) has been used at that point, check it again and undertake some tests on the variables. It could be something like:

At the first point:
Check how much money and time is used and put them into variables.

At the second point check those variables again and do a check like:

```
If((Δmoney > 1000€) && (Δtime > 30 days)){
        raise an error;
} else if((Δmoney > 500€) && (Δtime < 15 days )){
        raise a quality check;
}
```

This is one example. With two variables representing two resources check. We could do that kind of check on many different points for one check. For example, we could check money and time at the first point, money at the second and time at a third point then do the tests. [23]

3.4.1. Graphical representation of constraint

There are 2 issues with graphical representation:

1. It takes space on a diagram
2. It has to be easy to understand at a glance or else it loses its point.

There are 2 approaches to these problems. The first one is to make the graphical representation as tiny as possible. The problem is that it becomes difficult to understand if it is too tiny. Another solution is to find more space so we can still retain a detailed representation.

I think they are all equally inadequate. Sometimes we could make it really small without missing anything; other times, it was impossible to have something simple and representing what we wanted. That is why we think it would be better to combine both.

We would like to combine a quite simple representation in the Activity diagram like in Heidelberg [23] as shown in Figure 3-1

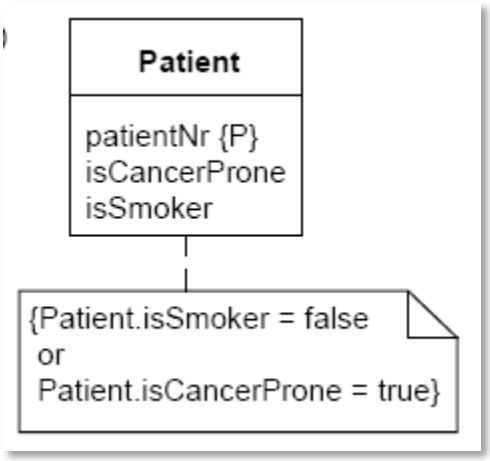

Figure 3-1 Simple note to show a constraint

It is a note to a single Class but we could use this technique to connect with many Nodes in our diagram. Something like Figure 3-1.

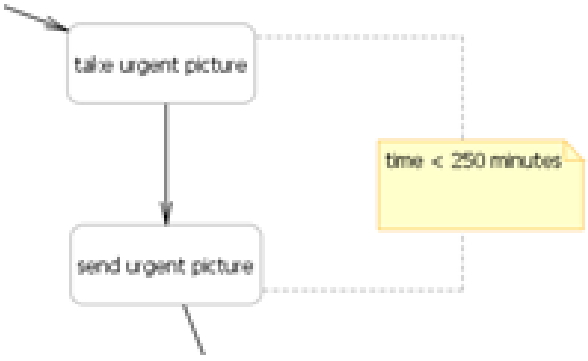

Figure 3-2 Simple note between 2 Actions

This constraint would describe that we want to check if the time between the start of the first action and the end of the second is less than 250 minutes.

It could be enough for this constraint but it will not always be that easy so we want to add a level to the constraint and open for the possibility to give more details to the constraint. We think the best way to do so would be to define the constraint with properties. Something that would look like Figure 3-3.

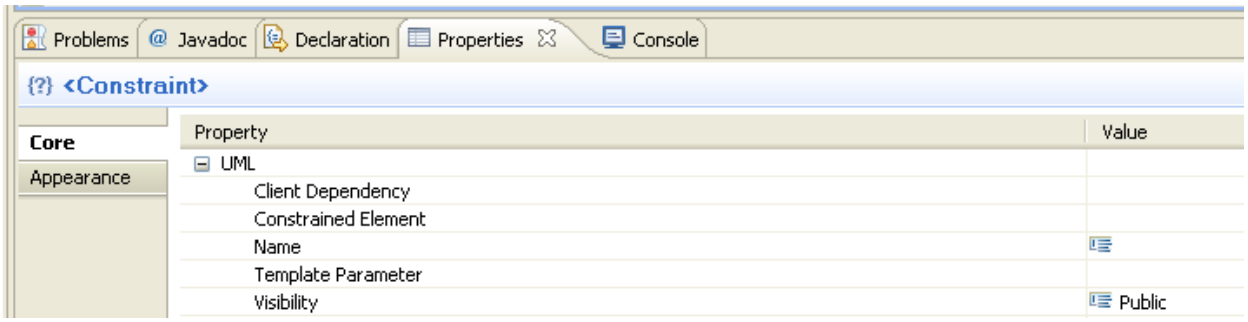

Figure 3-3 Constraint properties

We will have to define every possible property but we think it is the best way to define a constraint in a way that will not take too much place on the diagram and still be very complete.

4. Creation of the Tool

This tool is created to allow project control with Activity Diagram. We used some concepts from Model Driven Engineering to create it as we are using UML diagram that we transform to code. And to do so we used tools like Eclipse and GMF to achieve this. Let us see what Model Driven Engineering is about and how we used Eclipse and GMF to achieve this. After that, we will look into the logic of the tool and the challenges we met.

4.1. Model Driven Engineering (MDE)

Over the past five decades, programming languages has evolved quite a bit from first and second generation languages in terms of raising the level of abstraction, allowing developers to focus on the design intent rather than the underlying computing environment. The more recent advent of more expressive object-oriented languages like C++, Java and C# has raised the level abstraction even further. Furthermore, the use of today's reusable class libraries and

application framework platforms allows developers to reuse program code and domain specific middleware services. These advances helps developers create more advanced applications, as they can focus on the domain of the application, and do not have to reinvent the wheel each time.

A downside of the growing complexity of today's systems, however, is that it is hard for anyone to keep a full overview of a system as its complexity and the amount of implementation code grows. The use of complex middleware platforms, like J2EE, .NET and CORBA, containing thousands of classes and methods, in many of today's systems makes this even harder. Not only do these platforms have to be integrated and tuned with the domain application, but the complexity makes it hard for anyone to master them completely. Moreover, as these platforms, domain of the application, and/or the needs of a business often change rapidly, developers often spend considerable effort manually changing the application to reflect these new requirements or porting the code to different platforms. The effect of this growing complexity is that developers need to spend lots of effort on these implementation issues, rather than focusing on requirements and the domain of the application. The complexity of the systems also makes it difficult to know what parts of the system is affected by a change in the requirements or changes to the platform or language environment.

Another problem resulting from the growing complexity of computer systems and the rapid changes is that maintaining documentation requires a lot of effort, and is very time consuming. Also, since there is no direct linkage between the models and the implementations, there is a big chance that the documentation and the implementation will be out of sync – both due to changes during the initial development process and changes at later stages. This might also lead to that developers do not put in the effort needed to assure the accuracy of the documentation.

MDE addresses these issues, and aims at offering means to handle the growing complexity of these systems, and allowing developers to focus on and express domain concepts. This is achieved by combining the concepts of Domain-specific modeling languages and Transformations.

4.1.1. Domain-Specific Modeling Languages and Metamodels

Domain-specific modeling languages use a type system that formalizes the application structure, behavior and requirements within a particular domain. These languages are described using metamodels, i.e. metamodels describe the abstract syntax of the domain specific languages. This is achieved by describing precise relationships between the concepts of a domain, thus formalizing a language which can be used to describe specific domain related concepts. The domain specific languages can then be used to describe applications using domain-specific concepts instead of concepts of a more general purpose modeling or programming language. This means that developers can focus on the domain which they are describing, rather than on a specific platform, thus raising the level of abstraction.

On the other hand, there are general-purpose modeling languages. It is the opposite of Domain-specific. These allow a full range of semantic expression. One example of a general-purpose modeling language is UML. The UML metamodel describes a very general domain specific language for software development, and thus provides a common language that can be used by software developers to describe applications and business logic. A less general example could be a metamodel describing the relationships between a set of different concepts in the banking industry, providing a language to describe bank related concepts on a high level of abstraction.

Figure 4-1 shows a simplified metamodel for a class-model. This model describes the class-model domain, i.e. it describes the properties of the concepts comprising such a model, and the relationships between them. Thus it provides a formal language in which these concepts can be described.

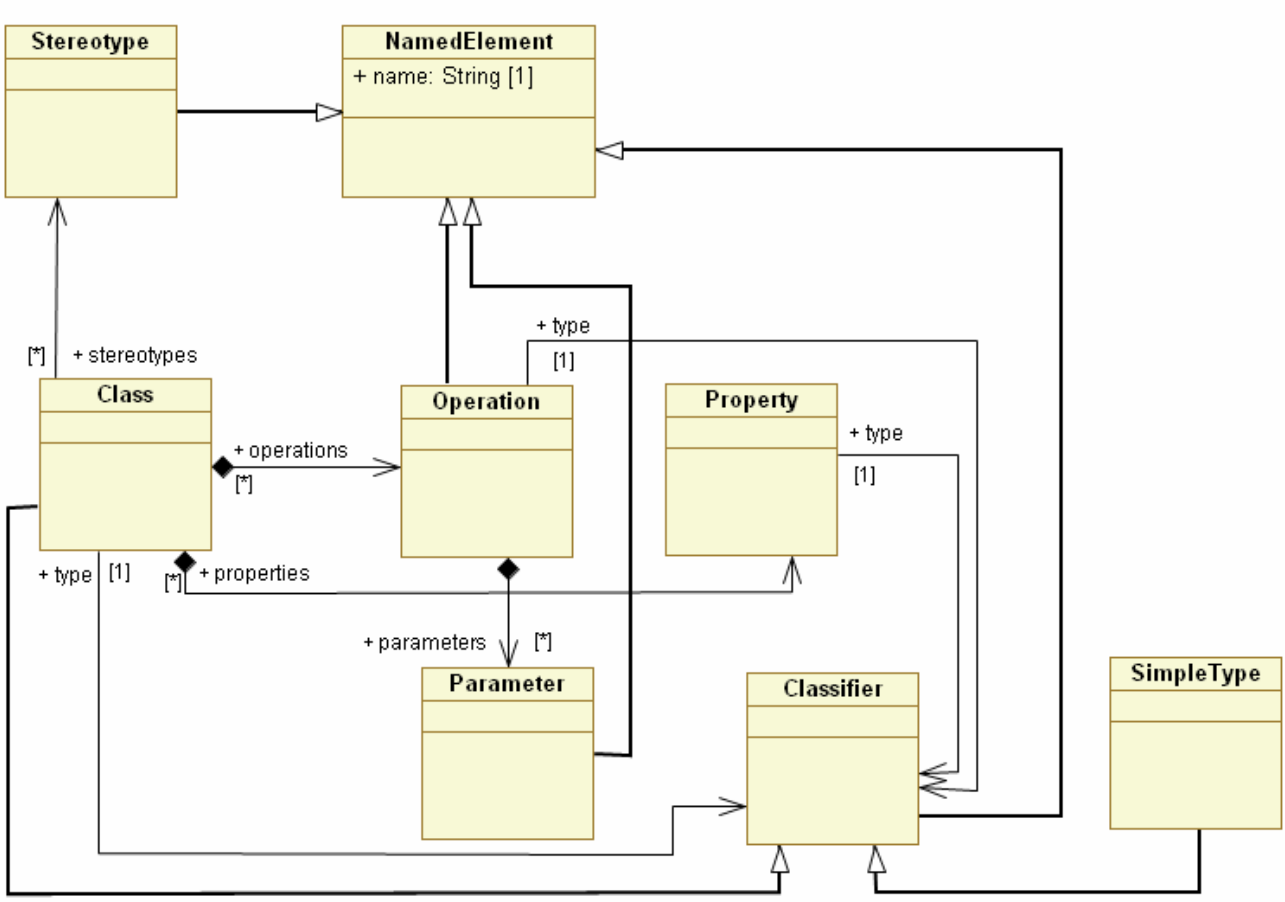

Figure 4-1 A simplified metamodel of a UML class-model

4.1.2. Meta-levels

The Object Management Group (OMG) defines four metalevels (Figure 4-2). These levels represent four levels in which a model may reside:

M0 - Contains the runtime instance of a model, e.g. the representation of a model element in a running application.

M1 - Contains the model, e.g. a UML model, which may be instantiated at the M0 level.

M2 - Contains the metamodel, e.g. the metamodel in Figure 2, describing the language used to define models at the M1 level.

M3 - Contains the meta-metamodel, i.e. the model describing the language used to describe a metamodel (i.e. a model residing at the M2 level). An example of such a model is the Meta Object Facility (MOF)[15], which is the language describing the UML metamodel. As models at this level are general enough to describe the concepts of languages used to describe other

languages (i.e. metamodels), they will also be general enough to describe themselves. There is hence no need for an M4 level.

The four metalevels and their relationships to each other are illustrated in the figure below.

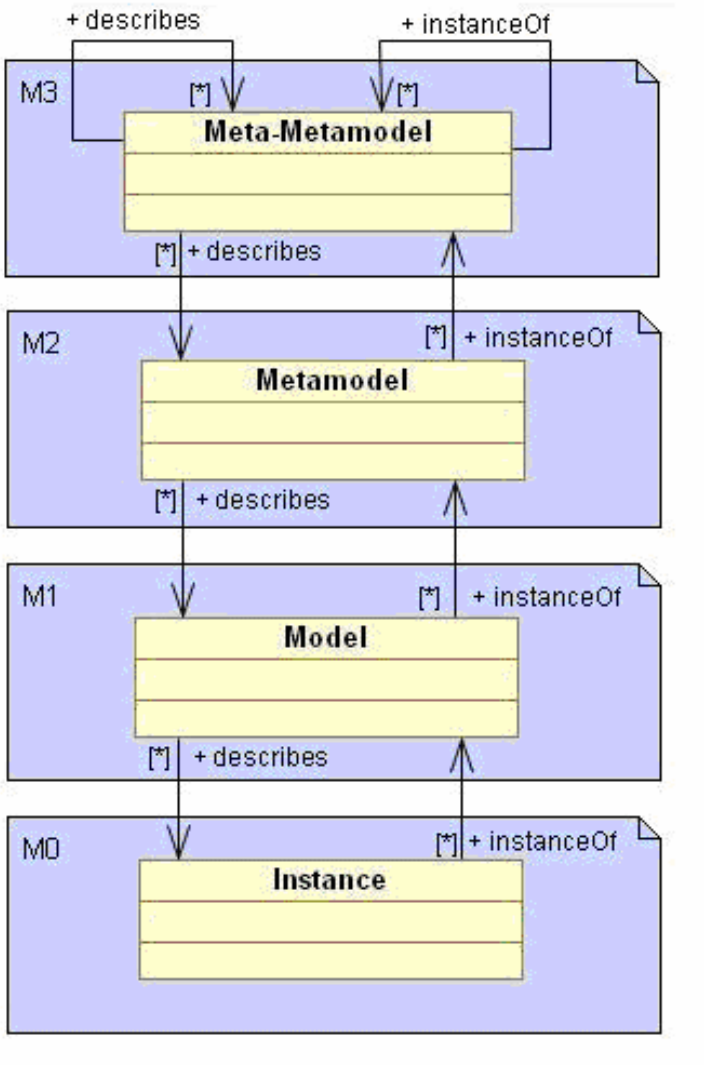

Figure 4-2 The four metalevels of OMG

4.1.3. PIM and PSM

Two terms that are often used in MDE are Platform Independent Models (PIMs) and Platform Specific Models (PSMs). A PIM is a model that is independent of the implementation technology. Thus, it does not have any information regarding the technology used to

implement the system – it describes the logic of the system. A PSM on the other hand is, as the name implies, a model describing an implementation of the system (or parts of a system) using a specific technology. Hence a PSM is a refinement of the PIM.

The purpose of creating a PIM is to allow developers to capture the details of a system without having to dig into the details of a specific platform or technology. Thus it allows developers to focus on the business logic rather than how a system should be implemented and what technology to use. It is a means to raise the level of abstraction. Besides of making it easier to keep the focus on the right things during the development process, PIMs also makes it easier to keep an overview and to get a better understanding of a system as all the details required to describe implementation issues are hidden away. Not only is this an advantage for developers during the development process but it also makes it easier for people without technological background or knowledge of a specific technology to understand the system.

4.1.4. Transformations

Transformations has the potential to help ensure the consistency between different models, documentation, requirements and implementation code as the evolution of artifacts are formalized by rules, and can thus be automated. For this to be possible the source artifacts of the transformation need to be precisely defined by e.g. a metamodel so that the artifacts might be processed by a computer. There are mainly two kinds of transformations used in MDE; model to model transformations and model to text transformations.

Transformations are described by mapping rules. A mapping rule is a formal description of the relationship between the input and the output of the transformation, i.e. it describes how the output is created based on the input. The input and output artifacts might be a single artifact or it can be a collection of artifacts. Also the result of the transformation might be computed based on the properties of the artifacts and/or the relationships between the artifacts that comprise the input of the transformation. The output might be created based on a metamodel or just as artifacts without any particular syntax. The first approach is typically used to describe model to model transformations, while the latter is typically used to produce text.

Using transformations between all stages in a software engineering process therefore makes it possible to automatically generate all the artifacts needed, based on one or more source models. This does however require mapping rules for each of the transformation steps in the process and the existence of metamodels formally describing each of the models.

4.1.4.1. Model to Model Transformations

A model to model transformation is the process of creating a model based on another. More precisely a model transformation creates instances, based on precise definition of the relationship between instances of one metamodel, and based on instances of another. The set of rules that are used to describe the transformation is often called a mapping. The model2model transformations can also be controlled and/or modified by parameters to the mapping itself. This means that the transformation may not merely depend on the original model but also on other things defined as parameters to the transformation.

4.1.4.2. Model to Text Transformations

Model to text transformations refer to the act of generating text based on information from one or several models. This is performed in a similar way as model to model transformations, with the exception that only the input of the transformation (i.e. the source model) is defined by a metamodel. The output is just informal text. In MDE, model to text transformations are mostly used to automatically generate implementation code from the models, but it might also be used to automatically generate documentation like java doc from the models. There are however many cases where not all the text can be generated automatically, but many parts of it, at least the skeleton of it can often be generated. The degree to which the process may be automated depends on the level of details of the model(s) used to generate the text, and how much logic one wants to encode in the transformation.

Generation of code is often performed based on the PSM, as the PSM describes the application in a platform specific way, e.g. an EJB implementation. In some cases however one can transform the code directly from the PIM, as it might be quite close to the actual implementation.

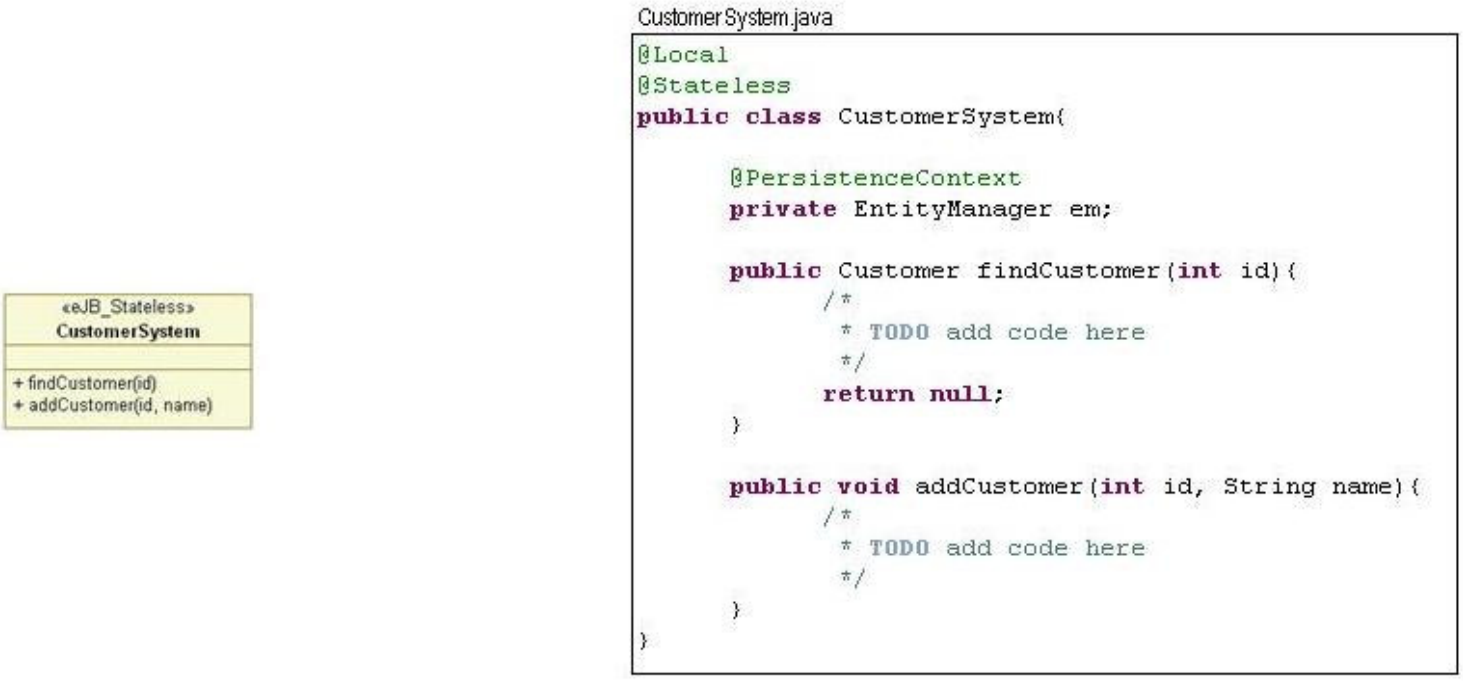

Figure 4-3 Illustration of the model to text transformation of the class CustomerSystem

Figure 4-3illustrates an example of model to text transformation.

4.2. How Do We Create an Eclipse Plug-in That Will Respect MDE Concepts

After what we learned about MDE in chapter 4.1, we want to create a plug-in that will stretch toward some of these MDE concepts. This because we are starting from an Activity Diagram that represent our PSM and we want to produce code that will handle API calls.

We decided to use Eclipse platform to do so. To achieve MDE on this platform, we first need to have all the pre-required plug-ins installed. As we are using UML to draw our model, we must have a tool that can draw diagrams and transform them to code. We used GMF to create our Activity Diagram and MOFScript to transform the diagrams to Java code.

The Graphical Modeling Framework (GMF) is a framework for building modeling-like graphical Eclipse-based editors. Examples of such editors are: UML editors, ECore editors, business process editors, flow editors, XSD editors, etc...

The framework can be divided in two main components: the tooling and the runtime. The tooling consists of editors to create/edit models describing the notational, semantic and tooling aspects of a graphical editor as well as a generator to produce the implementation of graphical editors. The generated plug-ins depends on the GMF Runtime component to produce a world class extensible graphical editor.

The Figure 4-4 briefly presents the workflow used to create a graphical editor using GMF and then focuses on presenting the benefits and features of the runtime component of GMF.

Although pictures in this article were taken from screenshots of a GMF Runtime client implementing a UML2 modeling application, the use of the GMF Runtime is not confined to the modeling space since any application that displays and/or edits the graphical representation of EMF models can leverage the framework.

Developers who create graphical modeling-like editors using GMF follow this simplified workflow:

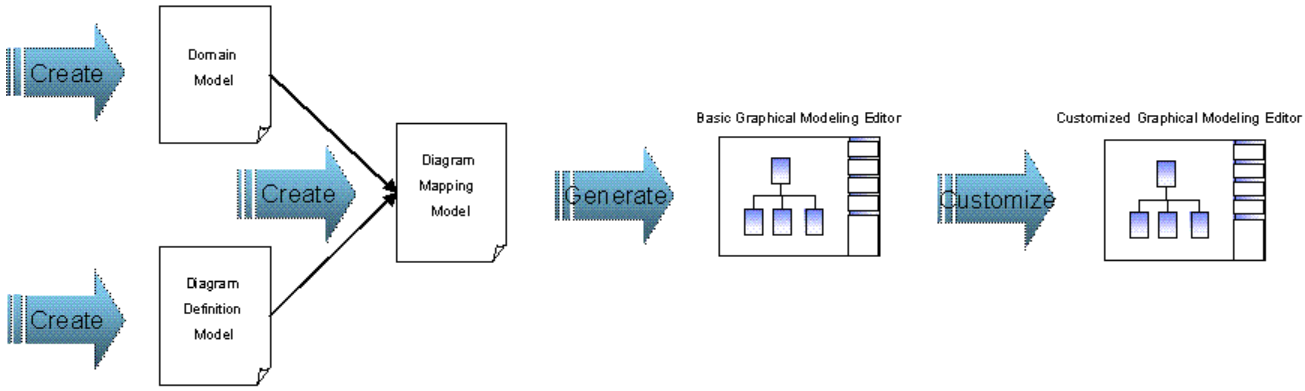

Figure 4-4 The Workflow in GMF

Steps are:

- Create a domain model. This model defines the non-graphical information managed by the editor

- Create a diagram definition model. This model defines graphical elements to be displayed in the editor
- Create a diagram mapping model. This model defines mapping between domain model elements and graphical elements
- Generate the graphical editor
- Enhance the graphical editor by editing the generated plug-in code

4.3. Challenges

Challenges are problems we may encounter with MADE. Here we will list some of them that we encounter. Most of the problems we discuss here are low level problems due to the architecture of Activity Diagram and Java code. Let us explain some of them.

The first challenge we encountered was that we needed to combine the UML.ecore with add-ons for representing our extra variables and our tests. The UML.ecore is the EMF representation of the UML meta-model created by the UML project at the Eclipse foundation. Since we wanted to represent constraints and resources that are not in the UML meta-model, we needed to find a way to add them. We saw two possibilities to do so. Either we could add it to the model (directly to UML.ecore or through profile) or we could use something already in the model that could suite our need. For example, there may be a possibility to use pre-condition and post-condition to represent our Runtime Constraints. The advantage of using these conditions is that we do not have to work on the UML.ecore or create a profile but the disadvantage is that we cannot use the pre- and post-conditions in the diagram as they are planned for.

After constraints and resources are added, we needed to get our MOFScript to recognize the add-ons. As it is this MOFScript that will generate code from our diagrams, we needed to tell MOF that it must look into the extra or adapted model. In MOF, there is support for multiple models so both methods mentioned at the beginning of the chapter are possible to use.

The challenges we talked about now are related to the tools. They are due to GMF and MOFScript. Now let us look at other issues that are general problems. The third problem was that Activity Diagram can show multiple threads running parallel. When the trace meets the

Join Node, we need to wait for every parallel thread to finish before we move on. As we chose to transform Activity Diagram to Java code, we will have support for creating new threads in the code. The challenge is the handling of the different threads. Since we chose to use Java variables to save the values we will check in Runtime Constraints, we need to handle variables in each thread that we will check later. We are using those variables to keep record of our Resources and there can be problems if, for example, the variables are accessed at the same time or if threads overwrite each other. The workaround we will have to use is to avoid those kinds of problems by creating local variables and handle changes when we meet the Join Node.

Another challenge due to threading is how we will check our constraints.

One important challenge we met is when meeting merging points (Join Node or Merge Node). It is important to know if there are more edges to come. When we follow the trace to a Join Node, we must know if every other edge has arrived so we can move on. If not, we must wait for all parallel "threads" to complete before we can move forward.
It is a bit different for a Merge Node. Here, it is only the code syntax that forces us to know if there are more edges. This is due to the order the Activity is traversed. When the MOF script "hits" a Decision Node, it follows one of the trace until it finds the Join Node then goes down the next trace and so on.

One last thing we need to mention here is the problem that took the longest time to resolve. It was GMF itself. Most of the time it works correctly and generates the Activity Diagram editor as expected. But some bugs in the GMF plug-in (or its dependencies) resulted in time spent on workaround and reviews of the models. Many problems of that kind were resolved by restarting the Eclipse platform or creating a new project and copying the models from the old one. As discussed in this paper, GMF is not perfect. But it takes time to find out what is wrong when it is in the GMF generation or GMF runtime.

In this project we used almost a year to get things working with GMF. It got off to a bad start after its initial release. There were lots of problems with the update site, so there were errors within the installation of GMF. We needed to remove everything and reinstall it manually. When, at last, we had everything installed correctly, we tried to find some documentation to get started. The only available resources were the Help pages within the Eclipse platform that

were sending us to the GMF tutorials. It was, and still is, a web page but there were many links that were wrong. After a while, the people working for the GMF project fixed the tutorials so we could get started.

We went through the MindMap example available there and it worked quite smoothly. There were examples to show us how to use all the wizards within GMF to derive all the models needed to create the MindMap plug-in. It was a success. So we tried to change some elements to test if we understood some details correctly. Small changes in the Domain Model for example. The MindMap example still worked but not very smoothly. The derived models included errors that we needed to correct manually and the plug-in generator ran much longer to create the plug-in code. We found out quite early that it was not a good idea to use the wizards to derive models and we started to create them manually. It went much better.

After a while, we had a good understanding of the GMF and started the creation of our MADE plug-in. It was a total chaos. The plug-in generators took hours to create the code or went out of memory in the process. There were actually thousands of bugs in the GMF plug-in. We worked hundreds of hours to fix the generated code to get a working plug-in. Our main problem was that we needed to re-generate the code every time we made changes in one of our models.

It took us a very long time to get things to work and GMF 2.0 was released around the same period. Things went much better with GMF 2.0. We could make the changes we wanted and generate plug-in code that was working. The GMF project has improved a lot and we hope it will stay on that path.

4.4. Logic

We need to think about how the code will look like. There are a number of rules that we need to predefine.

1. The code generator will first need to create the needed variables defined by the parameters in the diagram. Those variables are the ones we will use to make our tests later on.

2. The code will start from the initial node and move along the flow to the next Activity Node. Then again, it moves forward along the trace to the next and so on until the final node.
3. Before “leaving” a node and before “arriving” at a node. The generator will check if there is information about a test or a change in some variables. We may need to emphasize that there can be changes in the variables even though there is no test at that same point. For example, we may need to update the time an Action took for a test that will be done later in the process.
4. We need to define a code generation rule for every type of node in the Activity Diagram:
    a. An Action must have the name of the method it will call. The code generation will get the name String and put it in the code.
    b. Decision Node will result in an “if-else” statement. If there are more than two outgoing flows we will use “if-else if” statement.
    c. Final Node is not converted into code. We conclude that this is the final “}” in the Java code.
    d. Fork Nodes are probably the most difficult part since we need to run every concurrent flow simultaneously. We will then create a thread for every flow and start them at the same time.
    e. Join Node is marking the end of the parallelism. When we reach a Join Node, we will have to wait for every flow (thread) to finish before we can move on to the outgoing flow.
    f. Merge Node will be “invisible” under code generation. There will not be any code generated from a Merge Node, it only marks the end of “if-else (if)” statements.

4.5. The transformation

We can now add our transformation. The process starts by finding the Initial node(s) and then following the flow from there.

If the transformation meets an Action, it will call the API with the name of the Action and move forward through the next flow.

If we are moving to a decision node, we create an “if… else if… else if… else” for the number of edges going out from the node. When it meets a merge node, it will close the “if” block. We need some information to create the condition. We chose to make it as a test on a variable that has the name of the decision node. The value it is compared to is the name of the edge going out from the decision node. This means that we need this variable set. It is usually done by an earlier call to the API. In our example, it is the “order” action that sets if the order is an urgent order or not.
If the flow meets a Fork node, our transformation will create a new thread for each outgoing edge. And when the flow meets the Join node, we need to wait for every thread to finish its work. To achieve this we use the join() from Java.
If we meet a constraint, the transformation creates an “if test”. If the test returns true then everything is normal and we move forward. If not, we add this test to our list of errors so the program can “remember” what went wrong.

When a node is reached and the code is generated, we go to the next node(s) and do the same checks to see which type of node we met and create the right code until we meet the final node. At the end, when all nodes are transformed to Java code, we need to create a report that we show to the user. As for now it is printed to the console, but we could create pop up or send email. We could actually send an email when a constraint is not met.

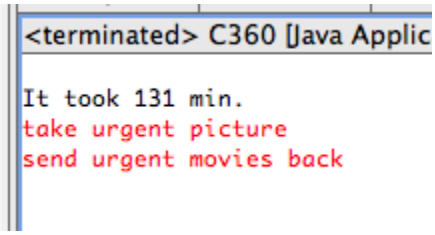

Figure 4-5 very simple report

Now we can draw an Activity Diagram as we want and add constraints. See Figure 5-1 for an example of an Activity Diagram. We have to remember that the name of the action will be the API call the program will make. When this is done, we can run our transformation and then run the program that is generated and see if everything works correctly.

5. Our Experiment

5.1. Introduction to Our Experiment

We will try to create a tool that uses Activity Diagram as a project management tool. The idea is that a project manager will be able to create a workflow represented by an Activity Diagram and add constraints to this workflow. From the activity, we will generate Java code that will check these constraints and give a report. To help us show what we achieve, we will now define a test case and some important concepts we will use throughout this case. We will then show our deductions. How and why we chose to create the tool as we did. Furthermore we will show what the limitations of the tool are as it is now and give some suggestions about what we could do further in order to improve the tool or adjust it for other purpose.

We want to show what the challenges of this kind of integration tool are and what the advantages would be. The hypothesis is that we can use an Activity Diagram to control a project. To show this, we will create the Eclipse plug-in that will make the connection between the UML Activity Diagram and the system for project control. After that, we will discuss what types of integration are available to us and why we choose one instead of others. In addition, we will explain what kind of problems there are and what solutions are available to us.

To simplify the lecture in this chapter, we would like to define some concepts that will be used:

- A Management Call or Manager Call is a call made by the project manager in the Activity Diagram. This is the name of the Action that will be transformed to an API call to the external system.
- Constraints in this chapter are runtime constraints. These are constraints that our project manager will add to the Activity Diagram. These are not UML constraints used in the validation of the diagrams.

5.2. Case Description

User: the person using MADE. She/he is considered to be the decision maker of the company. We will also call him or her a manager because the user is supposed to be a project manager in our case.

Company: The firm that gets orders from clients and delivers the finished products to them.

Client: the person or company that wants to get a 360 degrees film for his/her webpage.

Order: this is a job done by the company for the client.

Due-date: the date an order is to be finished.

Photographer: a person taking pictures.

Technician: a person assembling pictures in a 360 degree film.

Customer office: an office that handles commands from clients and deliver the product.

Working office: an office that consists of technicians only.

File sets: a file set is a bunch of files that correspond to an order. There is a one-one relation. This means one file set belongs to one order and one order can only contain one file set.

The company receives an order from a client.

The company creates an order in their systems. The order type is determined by its due-date. It can be normal or urgent.

The company assigns a photographer to an order.

A photographer takes pictures and sends them to a working office. The photograph does not decide which office he or she will send the files to. It is decided by the company on the background of the order type.

The working office receives the file sets. Employees at the working office are assigned to a file set.

When the employee has finished his 360 degrees film from the pictures, he or she sends it back. The files must be sent to the correct Customer office. This should be handled by the system to avoid errors.

When the files are downloaded to the customer office, the client should receive an alert.

The customer may want corrections of the 360 degrees film, the film is, once more, sent through the system as a new order (the old order is closed). The old files must follow the new order.

When the customer accepts the 360 degree film, the order is closed.

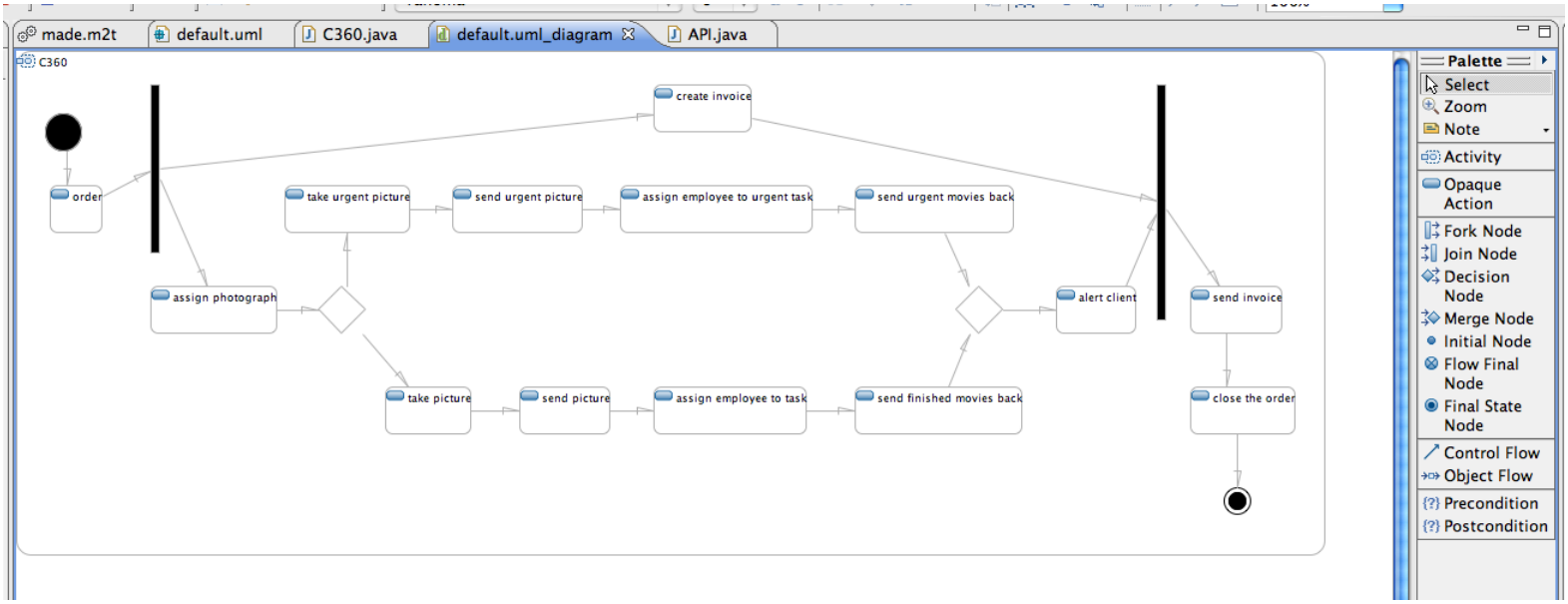

Figure 5-1 our example

5.2.1. Choosing a Type of Integration for the Tool.

There are 3 possibilities we can choose for the integration of the tool:

1. The first is to use direct API calls to a project control system. This means that every action in the diagram is Java (maybe other) commands of a program controlling a system. It would ask to start the next phase(s) when the diagram has finished an action.
2. The second API calls one or many software directly. So we would use MADE as a project controller. For example the first action would be to send a file, the second to send a notification of a file upload then the third would be a notification of download. We could think about extending these to use SOAP or XML-RPC.
3. The third is to export the diagram to a project planning tool and let the tool do the control. It would only be a project plan designing tool then.

There is also a fourth possibility that we did not mention here. We could create our tool with an API that other systems could call. This would simplify the creation of our tool as it will not need to depend on anything. The problem would be that the other system will need an extension or a plug-in to make those calls. We ruled this possibility away because we think it would be a great add-in but not good enough to be a standalone system. It is however a possibility we could look into in a later stage, after we have a working standalone application.

The first type (number 1) of integration has the advantage that we do know what type of call we are making. The API calls we will need are predefined in the project control system and we do not need to worry about calls to the system we are watching with our Activity Diagram in MADE. On the other hand, it will not let us have a very good control if something is missing in the project control system. For example, if we need some calls back to MADE that are not already in the project control system, it may not be easy to add them. Even impossible if the program is a closed program like MS Project.

The second type (number 2) gives us the type of control we do not have with the first. We can adjust MADE directly to the API of the system we want to control. In my opinion, it has both pros and cons. It is obviously good that we can modify our plug-in and it should work with other APIs. That will open MADE to almost any project. The disadvantage is that there is a lot of work involved in both, making the modifications, or creating MADE so it can recon all the API calls of a new interface that would be added. As mentioned above, we could create a SOAP service that would transform a WSDL to a new mapping of possible API calls. But this goes further than the scope of this paper.

The third type (number 3) is a very limited plug-in. If we develop this, it will only be a graphical representation of the project that will be added. We will also have to find or adapt an existing system that will allow us to export the project back to an Activity Diagram.

Now we need to decide which type we want to implement. If we compare the first and the second, we can see that both are about API calls. The first one restricts the calls to a project control system and the second plans to call any type of system. This means that the second one includes the first. So we can say that the second one is a better solution than the first.

Now if we compare the second and the third possibilities, we also think that the second includes the graphical representation. The third approach has the advantage that we could

use advanced project planning so we will not have to do it ourselves but we would need to find a way to communicate with those systems. In my opinion, it is better to keep control over every step between our tool and the systems used to perform the actions.

So we think that we should try to implement a tool that uses API calls directly to software or many different systems (Possibility number 2). We think it is better to show the possibilities in this part as it would prove that it can handle calls to a project control system (possibility number 1) since it would not be restricted to project control but would work with any system. Furthermore we also think it is better than the third possibility since we will not depend on other systems.

5.3. Analysis of the challenges

The solution we chose has, of course, some challenges. The first one we found is that we will have to introduce a restriction. For the tools to communicate together, the plug-in needs to know what the API calls are. We decided to use the name in an Action as the name of the API call. It creates the problem that it is the person creating the Activity Diagram that needs to know what the API calls are and write it as the name of the Action. For example: “order” as the name of the Action is valid because there is an API. This is a major restriction in our program. And we think that it is important to mention two ways to go forward with this problem. We could parse the API and restrict the name of Actions for the project manager. Or we will need to adjust the API to the calls made from the Actions.

Second, we need to have a two-way communication between our tool and the API because we want to create a tool that is not specific to one API. This means that our tool will make a call and needs a response to this call. The type of response will give us the level of control we can have in our tool. For example, if the response is only to say that the action is performed, the control we will have is very limited as we do not have any more information. If we want to use our tool to check the costs of a project, we will need the responses from API calls to return a cost. We could ignore this problem and report the result we would get to our project manager and open the possibility for him to fill the blanks. Another thing we also could do is to restrict the possibilities in the diagram to adjust the API. For example, you cannot create money constraints in the diagram if the API does not communicate any amount.

These two problems are quite alike. They create a restriction that can be solved by either adjusting the API to what you want to achieve with our tool or limit the possibilities of the project manager. So we will have to choose between more functionality and extra work for programmers or no additional work and limited functionality.

In our test, we will do the extra work so we do not limit the possibilities of the plug-in. But we cannot do this work for everyone. Therefore it will be important to look into how we can resolve these problems by limiting the possibilities for the project manager.

5.4. Tests Description

What we want to test is the usability of our plug-in. What does it offer that others do not? What are the limitations and what can we do to solve this. We will now show our first implementation and what changes we made to try to improve the tool and which inconveniences were added because of these changes.

In the initial experiment, we tried to create a tool made for managers without any IT knowledge. We thought it would improve the communication between management and IT department (or a computer based system). There are some problems with that goal. The main one is the communication between the two systems (our plug-in and the foreign system). We then decided to set a limitation that the Action should have the same name as API calls to the foreign system. This is due to problems when generating code. Since we need to call the API, we need to know what the name of the API call is.

We chose this solution since it did not involve rewriting the plug-in. The constraints are added on the user instead of our system. If we wish to allow a manager without any knowledge of the API to be able to use this tool, we will need to give a list of names he/she can use for an action. In my opinion, it will make the tool more difficult to create. We think it is a real challenge to improve this tool to check the API and give a list of possible names. And we will put this as a suggestion of improvement for the tool.

Let us have a look at the possibilities we have here. We can take for granted that the user of our tool knows what the API calls are and that he or she can use the calls correctly. With this assumption, we can go on with our plug-in as it is and assume that the communications are

correct. This would require a lot from the user but it will achieve our goal to allow him or her to add constraints to an Activity Diagram and check if the system fulfills the requirements. As mentioned above, an easy help for the user would be that our plug-in reads the API and gives a list of possibilities for the user to choose from. The plug-in could also check at runtime if the names written are correct API calls. We will not implement this at this point as the important thing is that the calls are correct and we will just assume that the user knows the calls.

Another possibility we should look into is what we should do if the call is incorrect. Our plug-in creates a Java code from the Activity Diagram and saves it in the Eclipse workspace. Eclipse, per default, compiles automatically after saving the Java file. It will then give us feedback with every error in the code. These errors will need to be corrected. We see two different approaches to this:

1. Remake the Activity Diagram to comply with the API.
2. Change the API so it allows more possibilities for the project.

Neither of these stand out as an obvious choice compared with the other. A further analysis will determine which to choose. For example, if the Management Calls are misspelled, the user will have to correct his or her diagram. In other situations, the Management Call is something missing in the API and should be added. Or another API would have to be added to fill in the missing part.
We see this happening in our test case. We can see that there is a part of the system that is handling the pictures and there is an Action that is creating an invoice. It will usually be two different systems that will handle those two things. So we would need two APIs. Let us try those two possibilities and see where it leads us.

We first tried to make it work with a minimum of changes to the API, so the first possibility was implemented. Adjustment to the Activity Diagram will be made to fit with the API. We did not make any changes to the diagram we have in Figure 5-1. The only thing is that the API calls are not planned to work with the runtime constraints the manager would want to check. For example, sending pictures does not yield a cost. It is probably a common problem for this kind of system as it only works to transfer the pictures. This part of the system has no knowledge of its costs. Another example is the time used. We can check the time used to send

the files but this is not the goal of the API call. We would have to add something to handle the time check.

So, maybe, the second solution must be used. We try to adapt the API so the management calls are correct and we can get some values we can use for the runtime constraints. From a usability standpoint, we should not change the API itself as it will slow down everything else using the API. The solution we chose was to create an add-on to the API. For example, if there is a money constraint, we create an add-on that will add money as a variable and call the API. Let us look at our example. When the manager call is made to send the pictures and we need to get a cost out from this call, we need to create the add-on that will call the API to send the pictures and added a system that will yield a cost for it. However, there is no simple solution to how this should be done. There are different possibilities like user input where we will ask the user to put the cost in the system, calculations where our manager would have to define how the costs are calculated, using another API that would ask the accounting system, etc…

With that kind of solution, we would need a great deal of work every time a variable used as a runtime constraint is changed. But, as mentioned before, we need to create a two way communication between the Activity Diagram and the APIs in use. In our test case, we moved forward with this hypothesis and we created this bridge between the Manager and the systems in use.

5.5. Results of the Experiment

Our experiment shows that we can create a tool that transform Activity Diagrams to Java code that can make API calls. However, to make a useful program, we had to write an interface between the Manager calls and the API. This interface adds functionality to the API or combines different APIs to handle runtime constraints. As mentioned earlier, the main problem is that APIs are not created to do anything more than what is expected from it in the system. For example, sending files is supposed to send files and nothing more so we needed to add something that will allow us to check the runtime constraints such as money or time.

6. Discussion

6.1. Reasoning behind choice of Tools to create MADE

There are two main points when talking about how we wanted to create our tool. We wanted to make it open source and we wished to use the MDA approach.

Open source software (OSS) can be defined as computer software for which the human-readable source code is made available under a copyright license (or arrangement such as the public domain) that meets the Open Source Definition. This permits users to use, change, and improve the software, and to redistribute it in modified or unmodified form. It is often developed in a public, collaborative manner. OSS is the most prominent example of open source development and often compared to user generated content. It generally helps the software develop faster and hopefully better than closed software. We choose this approach because we want that anyone interested in MADE to be able to use it and/or change it so it suits the needs of other users.

Model-driven architecture (MDA) is a software design approach launched by the Object Management Group (OMG) in 2001. MDA supports model-driven engineering of software systems. It provides a set of guidelines for structuring specifications expressed as models. The MDA approach defines system functionality using a platform-independent model (PIM) using an appropriate domain-specific language. Then, given a platform definition model (PDM) corresponding to CORBA, .NET, the Web, etc., the PIM is translated to one or more platform-specific models (PSMs) that computers can run. The PSM may use different Domain Specific Languages, or a General Purpose Language like Java, C#, PHP, Python, etc. Automated tools generally perform this translation. MDA principles can also apply to other areas such as business process modeling where the PIM is translated to either automated or manual processes. OMG focuses Model-driven architecture on forward engineering, i.e. producing code from abstract, human-elaborated specifications. OMG's Analysis and Design Task Force (ADTF) group leads this effort. The MDA model is related to multiple standards, including the Unified Modeling Language (UML), the Meta-Object Facility (MOF), XML Metadata Interchange (XMI), Enterprise Distributed Object Computing (EDOC), the Software Process Engineering Metamodel (SPEM), and the Common Warehouse Metamodel (CWM). Note that the term “architecture” in Model-driven architecture does not refer to the architecture of the

system being modeled, but rather to the architecture of the various standards and model forms that serve as the technology basis for MDA. The Executable UML (xUML or xtUML) is a specific approach to implement MDA.

So which tools can we use? The OMG released a white paper in February 2008 called "Understanding tool requirements for Model Driven Architecture" [24]. This paper explains some of MDA and makes a list of what the tool should be capable of.

Many tools claim to enable MDA development, but reality shows that only the code-generation part of MDA is supported. As a result, model-driven development has become a buzzword, the main objective believed to be the automation of 100% code generation. On the contrary, MDA should be used advantageously at various stages of the Software Development Life Cycle in order to enrich development methodologies and to enforce architecture conformance. The key technologies embodied within MDA tools should be focused towards enabling developers to maximize their activities during the MDA process.

For a tool to be MDA compliant, it needs to offer the user interoperability of all ingredients of the MDA process. So we shall define a MDA tool as a tool that provides meta-modeling, UML modeling, model-to-model transformations, model-to-text transformations, transformations execution and model repository.

Eclipse itself does not provide all these elements but there is a plug-in for each element. We use the EMF and UML plug-ins for modeling. There are ATL and QVT plug-ins for model-to-model transformation[25]. There is a MOF plug-in among many others like JET for model-to-text transformations[26]. There is also the possibility to include compilers for many different languages like Java, C, C++, etc.

So we use Eclipse because the platform architecture is based on the possibility to allow users to create their own plug-ins to do what they want and there is already plug-ins that makes it a good MDA tool. This is made easy because of the philosophy in the Eclipse helps other to add functionality. For example the GMF plug-in is a plug-in that helps creating plug-ins. There are many tools that simplify the creation of new plug-ins.

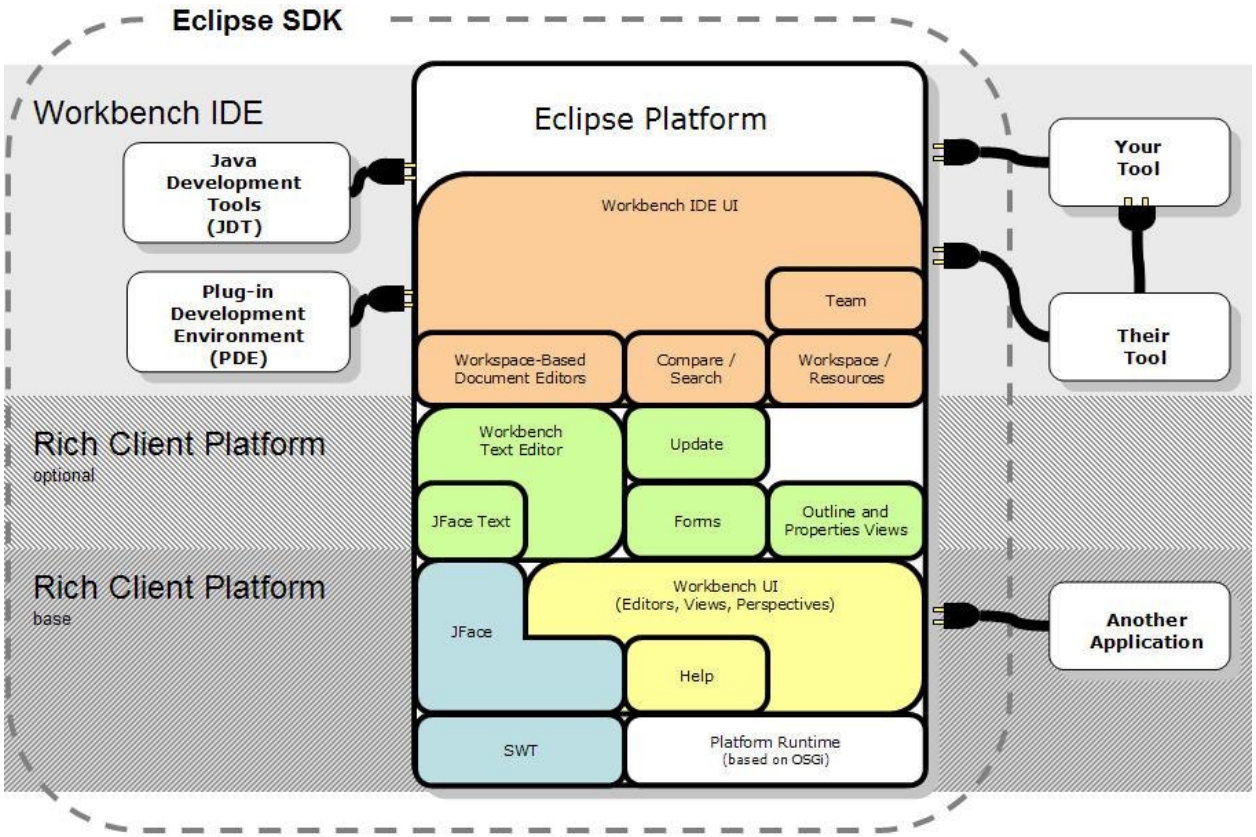

Figure 6-1 The Eclipse Architecture

The Figure 6-1 shows a plug-in that uses another one. This is exactly why Eclipse is so simple. We create MADE plug-in on the top of the GMF plug-in. And the GMF plug-in is created with other Eclipse plug-ins.

The second advantage of using Eclipse is that there is already a UML meta-model ready implemented in EMF. The UML meta-model is central in our work. At the base of the GMF plug-in, we need a meta-model that will represent what our plug-in will be about. And it is already done by competent people that worked really hard for a while to achieve it. It is certainly a much better model than the one we would have done ourselves. The only problem is that there is a complete UML meta-model and we only need the part about Activity Diagrams. So the model is way too big for us and it is not easy to get a complete overview of it. One of the main advantages is the GMF plug-in that is created for this kind of projects.

GMF is really good for us because a GMF user will get all the advantages of a generative approach – fully functioning diagramming code automatically generated and easily

regenerated upon model modification. Additional generator-specific variability points aid in customization of the generated code for the end-user. Generated code is optimized and proved by a large community. As many users will recognize, the generative part mostly involves working on developing a new language for describing diagrams and it fits perfectly into the Generative Programming[27] or Software Factories[28] concepts.

The other advantages are that it gives us a very good level of control since we can change models. We can add, remove or adjust elements in the UML.ecore (see chapter 3.2). This is the base of our plug-in. It is also open source, so we can change how our plug-in is generated by GMF. Furthermore it uses MDA/MDE[29], so it allows us to update changes really fast. For example, if we want to update the graphical representation of the Action node, we do this in one model and we can generate our new plug-in with this change.

GMF is starting to be a good tool but it was not like this when we started this project. When GMF came out in June 2006, it was working for small (meta-)models but could not handle bigger ones. For example, the UML tool project was using GMF to create their tools but had to re-write most of the generated code because it was wrong. Some contributors even wrote corrections to the code generator. At that time we needed to create the UML meta-model with many missing parts so that GMF could handle it. And there were still problems when the transformation was done. We really had big problems making it work. But we cannot put the blame only on GMF. At that time, it was Eclipse 3.2 that was the official release and many of the problems in GMF came from the Eclipse Platform. When Eclipse 3.3 was ready, GMF 2.0 had a much more robust platform to work on. The evolution of GMF since June 2006 is quite impressive. There were over 3000 bugs fixed for the last two years. By July 2008, there are still over 700 open bugs but the plug-in is working most of the time. Sometimes you need to restart Eclipse to get your changes visible but we did not encounter blocker problems since the release of GMF 2. Sometimes, there are things that are not working. However we can generally find workarounds to get things done even though it is still far from perfect. In the 2.1 release from June 2008, there were over 1200 bug fixes.

Why not use other tools? The main point is that we already knew the platform. We have been working for 6 years with a lot of different plug-ins and, now, we have a good understanding of the Platform. Without being experts, we know how to look for a plug-in that will suit our

needs. In addition, we can find out if there are problems with the system and resolve them or find help to resolve them.

We also tried to find other tools that are working with UML and code generation. There are quite a lot. After a short analysis, we found that around 90% are based on Eclipse. We can mention some like Rational, Omondo, Papyrus, MyEclipse, Poseidon and many more.

A last tool we ought to mention is Blueprint Modeling Environment (Blueprint ME). It is also based on Eclipse and they are supposedly the first one to support all the recommendations from the OMG for this kind of product. They actually integrate OMG's model-driven standards (MOF, UML, XMI, QVT and M2T). We would really like to see what it can do but it is coming only in summer 2008. So time limitation prevents us from analyzing this product thoroughly.

Therefore we will conclude that Eclipse is the best platform to build that kind of project on because most of the others are based on Eclipse and the OMG recommends the use of a tool based on Eclipse. In addition, there is the GMF plug-in that will help us to build our tool and we already have a good knowledge of the platform.

6.2. Discussion about the Results of the Experiment

Before talking about the result of the experiment, there is an important point to make about the experiment itself. We changed the models several times and often generated new MADE plug-in during this period. Once in a while we updated the platform (Eclipse, GMF and its dependencies) and we must say that it has been faster and faster every time. Of course we do the changes faster because we get better knowledge of GMF but also GMF itself was much faster to transform and run. The first transformation made took almost 2 hours and it took almost 5 more minutes to start Eclipse with MADE than without. After the last updates, changes and generation of the editor took less than two minutes and there is almost no difference when starting Eclipse with MADE or without. This is just to say that GMF has improved a lot since we started. Not only bug fixes, but performance as well.

Now, let us move on to the experiment itself.

As we said in Chapter 4.3, we needed to add constraints to the UML meta-model. We first tried to add them to the UML.ecore and it went ok but we moved from using the UML.ecore as it was too heavy for the GMF plug-in to parse. Then we created our own meta-model in EMF. It was a really simple one including only the element we used in our editor. Furthermore, it was much lighter for GMF so we moved on with this solution. We probably should add a profile to the UML to be more accurate. But time was against us. Mainly because we did not want to wait for 2 hours every time we made small changes and GMF did not handle big models very well.

As we moved forward in time, GMF became much stronger and could handle bigger models and we tried it as well. It worked well but it seemed quite irrelevant as there were another Eclipse sub-project called UML Tool that released Eclipse GMF based editors that were more complete and faster. We made some research about how they did this and found out that they actually use GMF to create code but there are some tweaks to the GMF generator to suit their needs. As it is open source, we tried to add our Constraints to the models they use without success because the workarounds they are using do not work for our Runtime Constraints.

After creating the Activity Diagram editor, we moved to code generation with MOFScript. We chose MOF instead of JET because the documentation is easier to understand and there were plenty of help to get.

The next step was to implement the logic we talked about in chapter 4.4. It was a straightforward transformation. We just followed the logic. The problem was to find how we could implement our Resource variables and runtime constraints. At first, we tried to define all variables at the beginning of the runtime then updated them as we moved along the traces in the diagram. When we created new threads, we accessed variables from the thread. This did not work because the result was wrong if we needed to test a Runtime constraint inside a thread. If we have parallel threads and there is a test to be made inside one of them, the changes to the Resource in another thread cannot be taken into account. So we changed this in order to create local variables inside every thread. It is the same principle as in the beginning of the Activity Diagram. We just create all variables locally in each new thread and update the main thread at the merge node. An important point here is that we need to wait for all the traces to reach the merge node before we can update the main thread. This is

because there still may be a Runtime constraint in a thread even after another one has finished.

Even though there are some difficulties within code generation from Activity Diagram with Runtime Constraints, we showed that we could create running Java code. Now let us have a look at some problems with the generated code.

We mentioned earlier that one of the main challenges is the communication between the Activity Diagram and the APIs we will call. The Manager calls usually expect more than what the API can provide to be able to check the runtime constraints. In our project, we are missing a reliable and complete way to add constraints variables to the API. We should be able to go further with this work to be able to do so. A possibility would be to combine with other diagrams, like Class Diagrams, to generate this interface that would allow us to add runtime constraints variables support to the API if the constraint variables are already supported. We could also analyze this problem the other way around and limit the runtime constraints to the API. In this case, we would have to check what the API offers define which runtime constraints we could use and limit the user to those variables defined in the API. This solution would restrict the usability for the manager. However, if we are going even further and accept that the API is actually designed for this kind of project, it should not be a big problem to adjust the API.

On a usability level, there are some points that should be enhanced to make this tool a usable tool. First, the plug-in should be extended to provide a list of possible Manager Calls to the user. One of the hypotheses in the beginning of this project was that we did not want to require the user to have knowledge of the APIs. The typical user was supposed to be project manager and possibly with limited IT experience. The solution to this would be to parse the APIs and create a list of all possible names the manager can select as the name of the Action in the diagram.

Another point is that we should add different warnings. In the current system, we give a report at the end of the trace. A user may want to be warned at once if something does not comply with his runtime constraints so he or she could adjust the workflow or change some constraints. To achieve this kind of live service, we should look into how we could send other types of warnings like email, SMS or API call to another system that will handle warnings.

To conclude this chapter, we need to summarize what we learned with our test. We can say that we showed that Activity Diagrams can be used for project control with a simple addition of Resources and Constraints to the meta-model. We could create a running Java code that checks our Runtime Constraints. There is no doubt that there is a need for further study to show how we could make the communication between the tools easier but we showed a way to do it. In addition, we also showed that Activity Diagrams can be used for project description, as a mean to communicate between management and development. It is not a new concept. In the early days of UML, one of its first uses was to help represent system requirements for a project. But we went further and showed that Activity Diagrams have their place in MDD. We did generate a running code from an Activity Diagram. As we discussed, Activity Diagram alone is not enough. It represents a workflow in the system and we would need more definition of the system. An idea that may be relevant is to create a class diagram first and then limit the name of the Actions in the Activity to the name of the methods in the class diagram. It looks like the overall architecture of the system may be quite complete when the code would be generated. But it is only a suggestion that needs to be investigated further.

6.3. Comparison

6.3.1. Is Activity Diagram suitable for Business Process Modeling?

Starting from a paper called Business Modeling with UML [30], the authors mentioned that: "Activity diagrams are often used in software development projects to document program flow, e.g., to show the algorithm used to implement a specific operation. Activity diagrams have a wide-range of uses, in that they can show activities (sequential and in parallel), the objects consumed, used, or produced by an activity, who is responsible for an activity, and the relationships and dependencies between activities. All of this is essential in business modeling." And also add that: "A process is simply a structured set of activities designed to produce a specified output for a particular customer or market. It implies a strong emphasis on how work is done within an organization, in contrast to a product's focus on what. A process is thus a specific ordering of work activities across time and place, with a beginning, an end, and clearly identified inputs and outputs: a structure for action. A process could in simple terms be described as a number of activities."

Additionally, we can mention Modeling Web Business Processes with OO-H and UWE [31] where the authors say: “In order to define the necessary constructs and modeling activities, we have decided to adhere to well known object-oriented standards, namely to the semantics and notation provided by UML. Using UML to model business processes is not new. From the set of modeling techniques provided by the UML, the activity diagram is the most suitable mechanism to model the business workflow, and so has been adopted by both OO-H and UWE to define the different processes. In this diagram, activity states represent the process steps, and transitions capture the process flow, including forks and joins to express sets of activities that can be processed in arbitrary order.”

The authors of these papers definitely believe that Activity Diagrams are suited for Business Process. But there are others like those who wrote the paper from ARC Discovery Project called On the Suitability of UML 2.0 Activity Diagrams for Business Process Modeling[32], where they make the point that Activity Diagram are missing some parts to be a complete Business Process Modeling tool. They offer comprehensive support for the control-flow and data perspectives allowing the majority of the constructs encountered when analyzing these perspectives to be directly captured. However, their suitability for modeling resource-related or organizational aspects of business processes is limited. It is interesting to note that they are not able to capture many of the natural constructs encountered in business processes such as cases and the notion of interaction with the operational environment in which the process functions. These are limitations that they share with most other business process modeling formalisms and reflect the overwhelming emphasis that has been placed on the control-flow and data perspectives in contemporary modeling notations.

The level of support observed for control-flow patterns illustrates that there is relatively broad support for capturing the various types of control-flow constructs that may arise in actual business processes. In terms of addressing the patterns that are not directly supported, we would like to make the following recommendations:

- Given the difficulties in capturing state-based patterns, most notably the interleaved parallel routing pattern and the milestone pattern, it may be worthwhile providing direct support for the notion of the place as it exists in Petri nets. Petri net places

capture the notion of “waiting state” in a much less restrictive way than the AcceptEventAction construct does;

- UML 2.0 Activity Diagrams currently do not support the creation of new instances of an activity while other instances of that activity are already running. This could be resolved through extensions to the ExpansionRegion construct in order to allow further instances to be dynamically created after the activity has started; and
- Given the lack of support for the synchronizing merge, a concept similar to the OR-join could be added to UML 2.0 Activity Diagrams.

As for data pattern evaluation, Activity Diagram offers a good support for this as well. Furthermore, the following remarks can be made:

- There is no notion of cases or distinct process instances in UML 2.0 Activity Diagrams, hence all data is effectively block-scoped by default and parallel threads of execution occur in the same data space. This could lead to some problematic situations when modeling highly data intensive and/or highly concurrent processes.
- The use of “tokens” as the fundamental underpinning for control and data flow introduces some subtle variations that do not exist in other Process Aware Information Systems (except those based on Petri-nets) – in particular data elements are truly consumed (and cease to exist) when they are passed to an activity for the duration of the activity. This also makes it difficult to actually share a data element/object between concurrent activities. On the other hand, it minimizes concurrency problems.
- The token approach provides an effective basis for internal data interaction. In particular, multiple instance data handling seems to be supported for all three multiple instance situations: designated multiple instance tasks, multiply triggered tasks (loops) and block tasks with a common decomposition.
- There does not seem to be any ability to model things “outside of the model” i.e. in the external environment. Hence there is no real ability to support external data interaction patterns. This may be addressed by using UML Activity Diagrams in conjunction with other diagrams such as UML interaction, overview and sequence diagrams, but this requires that the relationships between these diagrams are carefully established.

The resource patterns evaluation indicates that the support in UML 2.0 Activity Diagrams for the modeling of work distribution directives is relatively minimal. This reinforces the fact that UML 2.0 Activity Diagrams tend to be control-flow and data-centric and mainly aim to capture simple static routing directives associated with actions. They do not provide means for representing the subtleties associated with selective work allocation across a range work items at run-time. In particular, there is no real support for modeling any form of work distribution other than direct allocation or role-based allocation. There is no opportunity to utilize data resources (either within the model or externally from the environment) thus any opportunity for modeling organizational, history-based or capability-based allocation is obviated. Similarly, there is no support for specifying any form of work distribution algorithm or employing varying styles of work distribution (e.g. push versus pull, offer versus allocation).

Other observations arising from the resource patterns analysis include:

- The fact that the partitions can result in actions being simultaneously allocated to more than one resource can lead to difficulties where a means of providing role-based work allocation to a single resource is required. It is important to note that the resolution of this situation must be addressed as part of the implementation of the actions within the Activity Diagram.
- The ability to use OCL statements in the specification of partitions (and also for specifying relationships between partitions) would enhance the capability of UML 2.0 ADs to capture possible resource allocations, both in terms of precision and the range of work allocation strategies that could be represented.

Another paper called “Extending the UML 2 Activity Diagram with Business Process Goals and Performance Measures and the Mapping to BPEL” [33] by Korherr and List introduce the concept of adding functionality to the Activity Diagrams with profile to add some constraints to the diagrams. Here is the example they chose in their paper:

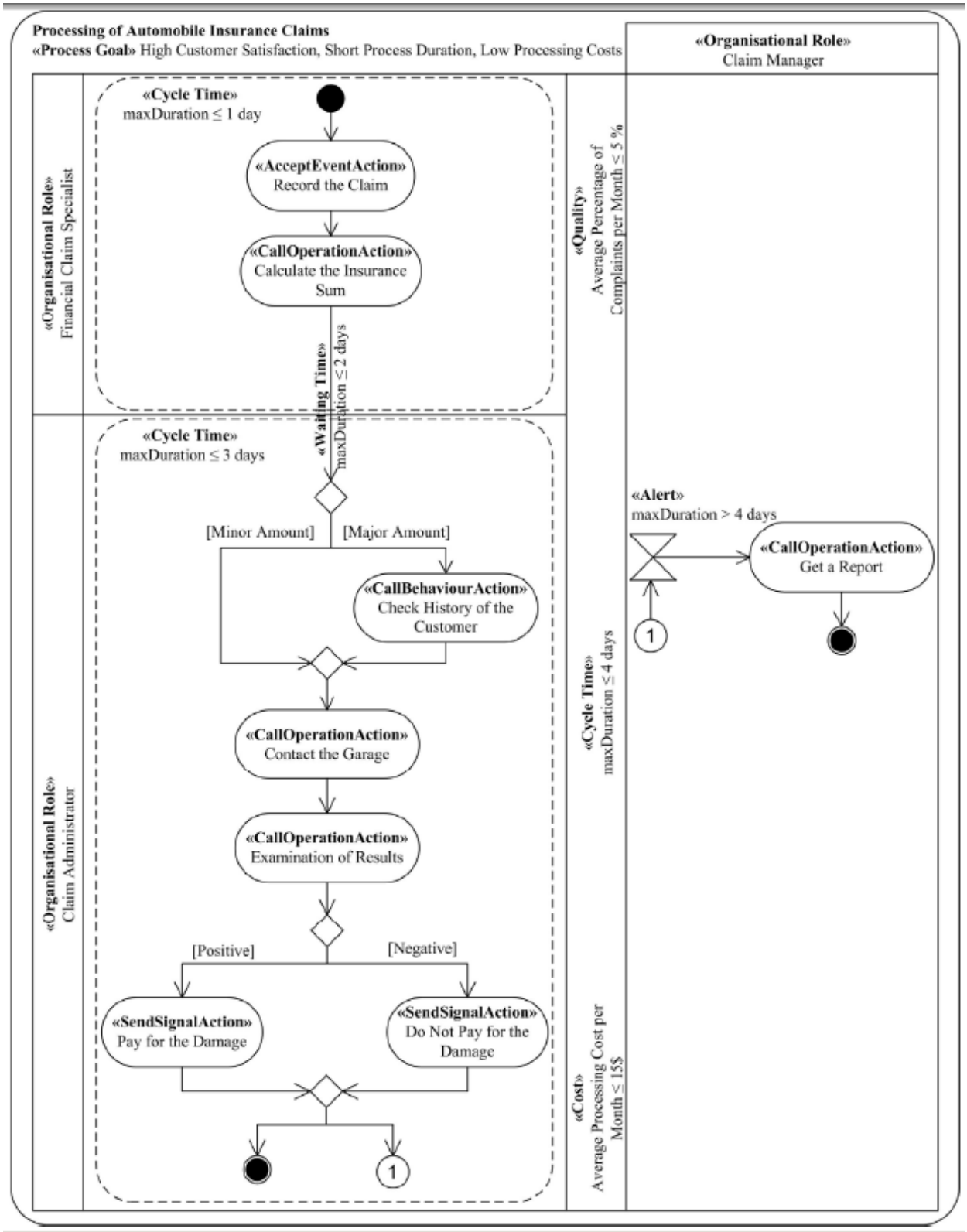

Figure 6-2 Activity Diagram with constraints

Their point is that Activity Diagrams are a part of the behavioral models of UML 2 and are used for modeling business processes as well as for describing control flows in software. Activity Diagrams neither have quality nor quantity based elements to measure the performance of a business process. For instance, the modeler of a process has no possibility to express the maximum time limit for the processing of a specific action - the basic element of Activity Diagrams - or a group of actions. And they solved this problem with Constraints added to the diagram.

The papers discussed above suggest that Activity Diagram as they are, are not good enough to represent business process but very close. Our conclusion is that we can add constraints to them in order to make them better and Activity Diagrams will then be useful to represent Business Process. We think we can conclude that our approach is a good one as we add constraints to the Activities to create some kind of control over the business processes for the Manager.

6.3.2. Why choose our tool and not the other?

As mentioned earlier, the other tools we have seen provide possibilities to draw Activity Diagrams and some even provide code generation. But our tool goes further to delegate control to the manager. We needed to add constraints to the Activity Diagrams to reach toward compliance with Business Process modeling. Some of the tools mentioned earlier have the possibility to add profile so we could draw Activity Diagrams with constraints although we could not find any that will allow us some code generation from Activity Diagrams with profiles.

So if you want to get the specific characters of MADE, you will need something more than what the other tools are offering. This means that someone who needs Runtime Constraints in Activity Diagram that will generate Java code cannot use one of the other tools as they are. This user will need MADE. He or she can, of course, create his/her own transformation to generate code in many of these tools.

Another point we need to make here is that, in some cases, we could use MADE in other programs like Rational. All software based on Eclipse and that allow plug-ins can run MADE.

Why Eclipse platform with GMF? Eclipse is a successful platform with a huge number of users and many developers. It is not only an advantage as we saw that GMF had many bugs and this may be caused by the number of programmers that needs to be coordinated. But bugs get fixed and it usually happens really fast. The other advantage is that the community is active and there is almost always someone that has a good suggestion or a solution to the problems you could meet. Eclipse and GMF have mailing lists and forums where you can get help. Through this project, there was never a problem we met that was not solved or has been worked around.

6.4. Generalization

There are three generalizations we should look into. The first one is Activity Diagrams as model for transformation from model to text that produces code that we can run. In my opinion, we did show that Activity Diagram is a possibility to use this concept. We created a program that is running from an Activity Diagram. Actually, we think it is one of the best diagrams in UML for model to text transformation since it represents active actions that can easily be translated as a running process or a simple method. We did specify this in our plug-in and that makes it easier although there are no apparent reasons that show that it would not work for any type of transformation.

After that, let us have a look at GMF. We used it to create an Activity Diagram editor but we could make it for any type of diagram. The UML tool project shows that we can make any UML diagram. And there are a lot of other projects, like mind mapping, that show that there are many possibilities. As long as you can create a meta-model of what you want to achieve, GMF can create a diagram editor. Even though, it can be really difficult because of the few predefined concepts. We are now talking about GMF concepts and there are only three. You can find Node, Link or Label. Even if a user should be able to represent anything with this, it can take a long time. In GMF, we need to define the graphic representation of everything as a Node, Label or Link and it is working. But some things as simple and common as a table are not so easy to represent. My guess is that it will come, sooner or later, integrated in GMF but, for the time being, it is not included. My main point here is that the tool is still young and

there are still some bugs and improvement that should be made to make it a standard used by most of us.

The last generalization is more of a usability point. We already mentioned that we used MADE as a tool to allow users to watch over a project and that, in our case, we needed to add an interface between the API and the generated code to add constraints that are usually not part of the system we want to inspect. For example the cost of sending files is usually not part of the API that actually sends the files. There should be a way to generalize this process and should not oblige programmers to add this manually.

7. Conclusion

We worked a long period on this project and we have seen many changes during this period and we can safely say that Eclipse and GMF have improved. The speed of this improvement is impressive and proves that the Eclipse community is solid, dedicated and has a really high standard. A relevant example is the speed of the development of GMF which is outstanding. It has advantages like how fast a bug is fixed and disadvantages like the number of bugs in the first release. We also have to keep in mind that the GMF development group was created only a bit more than a couple of years ago. This may also be a sign that this kind of tool is what the development community will move towards in the next few years. Eclipse is already a very well known and stable platform used in most of the tools we looked at and GMF will surely strengthen this position. On the other hand, we also noticed that Eclipse can be really heavy when it has a lot of plug-ins installed.

With our project, we showed that there is a possibility to create the kind of tool that can give extra control to a manager with the use of Eclipse and GMF. However we needed to add an interface between the API and the generated code so we could get some values for our runtime constraints. This is a disadvantage of our tool and something that needs to be studied. As we mentioned in this paper, the main challenge for this project is the communication between the tool and the API.

Another point the project has made is that we showed, once more, that it is easy to work with Model Driven Development as we only need to change the Activity Diagram and run the transformation to have the new program that we can run. This programming approach will probably be used more and more often when reliable tools will be available.

8. Future Work

- Two-way project management. Creating a list with possible Manager Calls and add the possibility to use forward engineering to create new API (new methods) in the system in use. Another possibility to do so would be to create the reverse engineering like a text to model transformation that would give a list to the manager of all possible API calls.
- Work on generalizing the creation of the interface between API and project control.
- Work with usability of this plug-in to make it user friendly.
- Work with other standards in order to create a better and easier interfacing towards API. For example creating a technology that would use Class Diagrams with Activity Diagrams to define APIs.

9. References

1. Project, E. *The Eclipse Modeling Framework (EMF) Overview*. 2006 11-02-2006 [cited; Available from: http://www.eclipse.org/emf/docs.php?doc=references/overview/EMF.html.
2. *Graphical Modeling Framework.* 2008.
3. OMG (2006) *UML 2.1 Superstructure spec convenience document*. **Volume**, 227-436
4. Ambler, S.W. *Agile Modeling - UML2 Activity Diagram*. 2006 [cited 2008 2008-05-11]; Available from: http://www.agilemodeling.com/artifacts/activityDiagram.htm.
5. *Eclipse*. 2008 [cited 2008 2008-04-06]; Available from: http://www.eclipse.org.
6. (2006) *Eclipse Platform Technical Overview*. **Volume**, 19
7. *Project management*. 2008 [cited 2008 2008-04-06]; Available from: http://en.wikipedia.org/wiki/Project_management.
8. Hunt, A. and D. Thomas. *The Pragmatic Programmers*. 2008 [cited 2008 2008-05-12]; Available from: http://www.pragprog.com/.
9. Rik Eshuis, R.W. (2002) *Comparing Petri Net and Activity Diagram Variants for Workflow Modelling – A Quest for Reactive Petri Nets*. **Volume**, 1-30
10. Ali, J. and J. Tanaka, *Implementing the Dynamic Behavior Represented as Multiple State Diagrams and Activity Diagrams.* Journal of Computer Science and Information Management (JCSIM), 2001. **2**(1): p. 22-34.
11. Eshuis, R. (2003) *Semantics and Verification of UML Activity Diagrams for Workflow Modelling*. **Volume**, 1-36
12. Bill Moore, D.D., Anna Gerber, Gunnar Wagenknecht, Philippe Vanderheyden (February 2004) *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework*. **Volume**,
13. *Model Development Tools (MDT)*. 2008 [cited 2008 2008-05-12]; Available from: http://www.eclipse.org/modeling/mdt/?project=uml2.
14. *MDT-UML2*. 2008 [cited 2008 200-05-12]; Available from: http://wiki.eclipse.org/MDT-UML2.
15. *MOFScript Home page*. 2007 [cited 2008 2008-02-16]; http://www.eclipse.org/gmt/mofscript/doc/MOFScript-Project-Description.doc, http://www.eclipse.org/gmt/mofscript/doc/MOFScript-User-Guide.pdf]. Available from: http://www.eclipse.org/gmt/mofscript/ http://www.modelbased.net/mofscript/.
16. LLC, G. *MyEclipseIDE*. [cited; Available from: http://www.myeclipseide.com/.
17. *PapyrusUML*. [cited 2008 2008-04-06]; Available from: http://www.papyrusuml.org.
18. *Package org.eclipse.emf.ecore*. 2008 [cited 2008 2008-05-12]; Available from: http://download.eclipse.org/modeling/emf/emf/javadoc/2.4.0/org/eclipse/emf/ecore/package-summary.html.

19. *Wikipedia*. Wikipedia 2007 [cited 2007 2007-10-02]; Available from: http://en.wikipedia.org/.

20. *Reources (Computer Science)*. Wikipedia 2007 [cited 2007 2007-10-02]; Available from: http://en.wikipedia.org/wiki/Resource_%28computer_science%29.

21. *Resource (Economics)*. Wikipedia 2007 [cited 2007 2007-10-02]; Available from: http://en.wikipedia.org/wiki/Resource_%28economics%29.

22. *Resource (types and developments)*. Wikipedia 2007 [cited 2007 2007-10-02]; Available from: http://en.wikipedia.org/wiki/Resource_%28types_and_developments%29.

23. Heidelberg, *Data modeling in UML and ORM revisited.* Proc. EMMSAD'99: 4th IFIP WG8.1 International Workshop on Evaluation of Modeling Methods in Systems Analysis and Design, 1999. **June 1999**(June 1999).

24. Goede, K.d. and J. Irizarry, *Understanding tool requirements for Model Driven Architecture.* 2008: p. 6.

25. *M2M*. 2008 [cited 2008 2008-06-01]; Available from: http://www.eclipse.org/m2m/.

26. *Model To Text (M2T)*. 2008 [cited 2008 2008-06-01]; Available from: http://www.eclipse.org/modeling/m2t/.

27. Eisenecker, K.C.a.U.W. *Generative Programming - Methods, Tools, and Applications*. 2000 [cited 2008 2008-05-31]; Available from: http://www.generative-programming.org/.

28. *Software Factories*. msdn 2008 [cited 2008 2008-05-31]; Available from: http://msdn.microsoft.com/en-us/architecture/aa699360.aspx.

29. *OMG Model Driven Architecture*. 2008 [cited; Available from: http://www.omg.org/mda/

http://www.omg.org/mda/specs.htm.

30. Hans-Erik Ericsson, M.P. *Business Modeling with UML*. **Volume**,

31. NORA KOCH, A.K., CRISTINA CACHERO AND SANTIAGO MELIÀ *Modeling Web Business Processes with OO-H and UWE*. **Volume**,

32. Nick Russell, W.M.P.v.d.A., Arthur H.M. ter Hofstede, Petia Wohed (2006) *On the Suitability of UML 2.0 Activity Diagrams for Business Process Modelling*. Research and Practice in Information Technology **Volume**, 1-10

33. List, B.K.a.B. (2006) *Extending the UML 2 Activity Diagram with Business Process Goals and Performance Measures and the Mapping to BPEL*. **Volume**, 1-12

Appendix

Abbreviations used in this paper

Abbreviation	Signification
EMF	Eclipse Modeling Framework
GEF	Graphical Editing Framework
GMF	Graphical Modeling Framework
MOF	Meta-Object Facility
JET	Java Emitter Template
PDE	Plug-in Development Environment
UML	Unified Modeling Language
MDD	Model Driven Development
MDA	Model Driven Architecture

The creation of MADE

- We create a GMF Project.
- We import the UML.ecore and UML genmodel from the UML plug-in to our project. We add them as ecore and genmodel to our GMF project.
- We add a gmfgraph to our project. We then create all the graphic representation of every node and links in the gmfgraph. There are two parts to this: define the graphic representation then associate the node or link with the graphic representation. There is a list of predefined boxes (rectangle, rounded rectangle, etc) and you can your own polygon. You can add shapes that you create with Java code or you can use the polygon shaping tool from GMF. You simply add the points representing the corners of the intended shape. You just add coordinates for each point in term of X (the horizontal distance from the center) and Y (the vertical distance from the center). The only thing you need to be careful with is

the order in which you define every point. The drawing utility will draw the points in the order you create them.

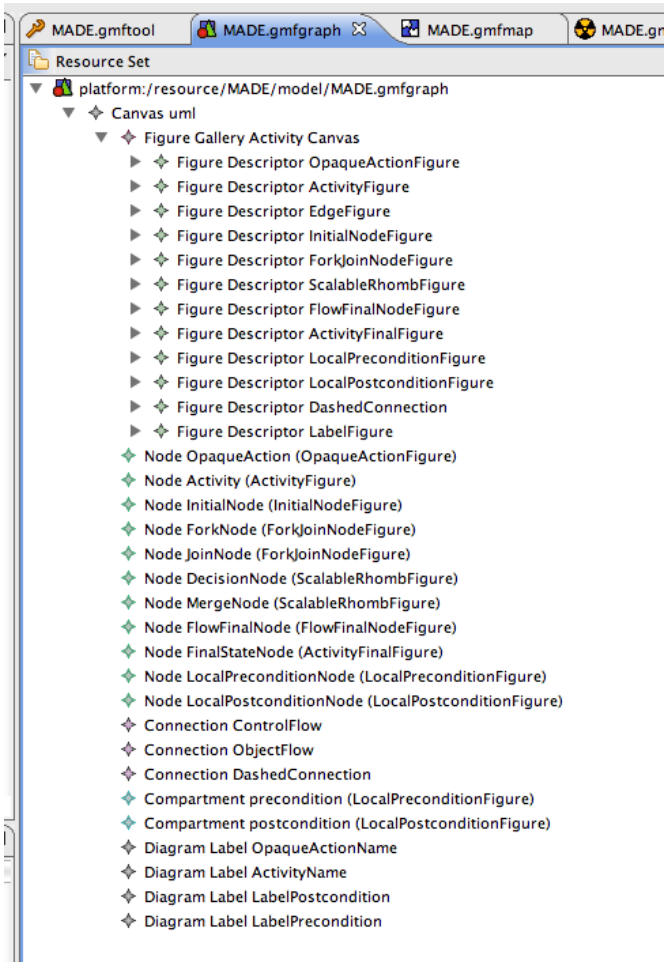

Figure 0-1 our gmfgraph

- We now add the gmftool to our project. This is the list with tools on the right that we use for creating a new Element in our diagram. This is called the palette in Eclipse. We create first every Tool Group. That is the groups that will appear in the palette. Each group is separated with a line in the palette. There is also the possibility to create drop down menu for elements. For example, if you want to have the decision and merge node to be in the same place in the menu, you can create the drop down where the user will be able to choose between the two.

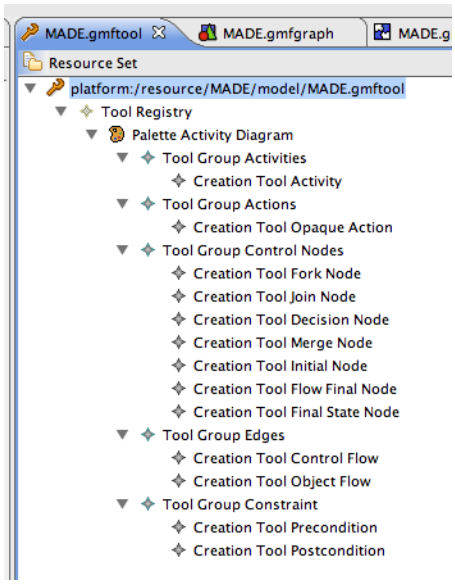

Figure 0-2 our gmftool

We, now, add the gmfmap to our project. There, we will map (associate) a UML element with a graphic representation and a tool. The association is important and is pretty straight forward. The challenge here is to have every Element on the correct level. For example, we need to have every Action inside the Activity. This means than the Action is on a level below the Activity. As well as the name of the Action is a level lower that the Action so it will be inside the Action. The Figure 0-3 shows that the Activity mapping has a Child Reference for every node that we will include inside our Activity Diagram.

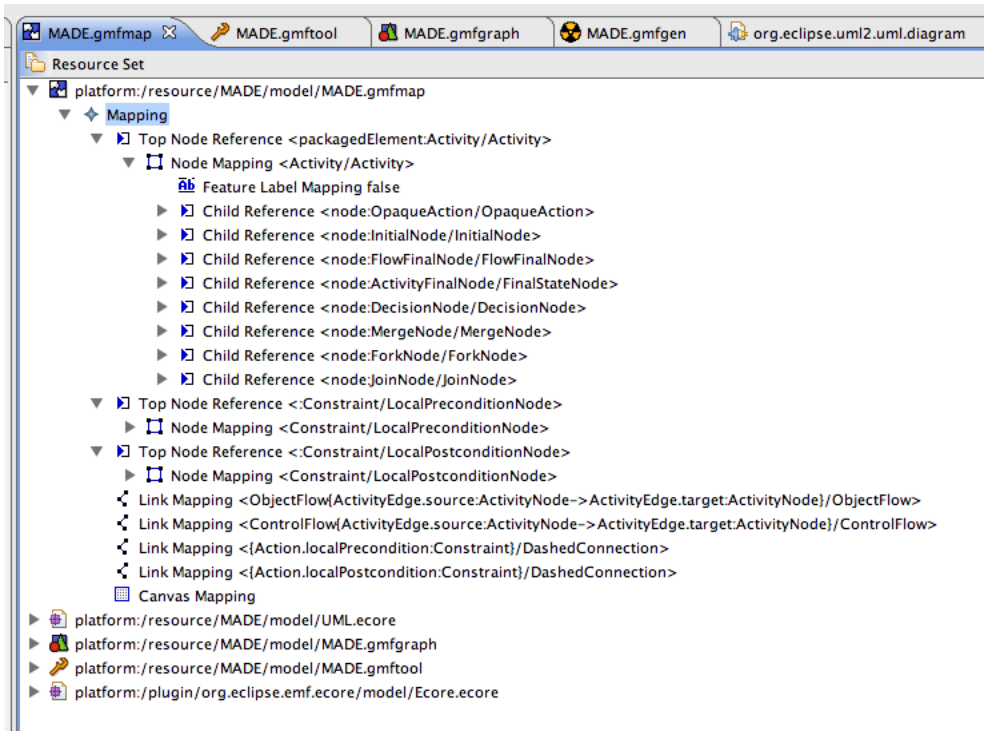

Figure 0-3 our gmfmap

- We generate the genmodel. We right click on the gmfmap and choose “create generator model…” in the list. We can make some changes here but it is not often necessary. We can change, for example, the name of the project that will be created from this model.

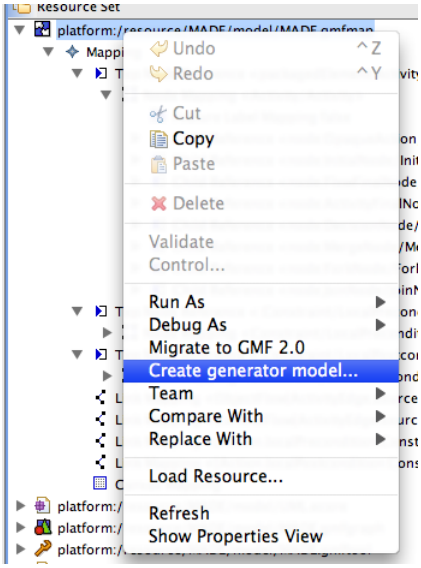

Figure 0-4 create generator model...

- Now we right click on the generated genmodel file and choose “generate the code” for the Activity Diagram editor. It will generate a new project inside our Eclipse workspace. This project includes all the necessary code to run our plug-in.

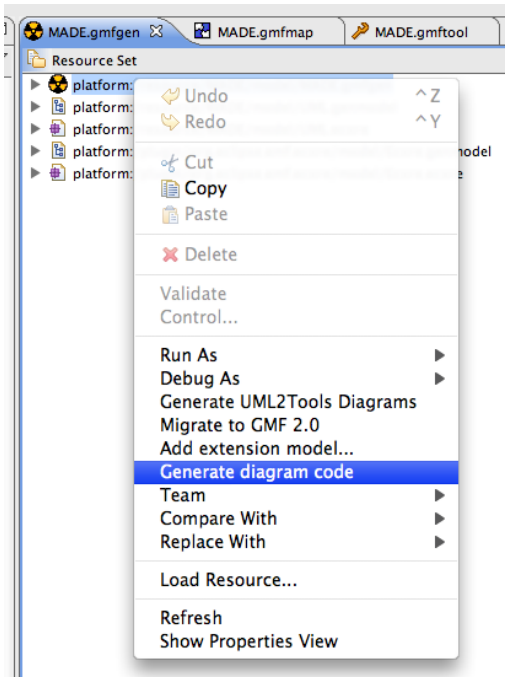

Figure 0-5 Generate diagram code

We can now right click on the new project and “run as” -> “Eclipse Application” and we have our Activity Diagram editor running as an Eclipse plug-in. This is the easiest way to run Eclipse with our new plug-in. We could, of course, export the code as an Eclipse plug-in and make Eclipse use it when it launches.

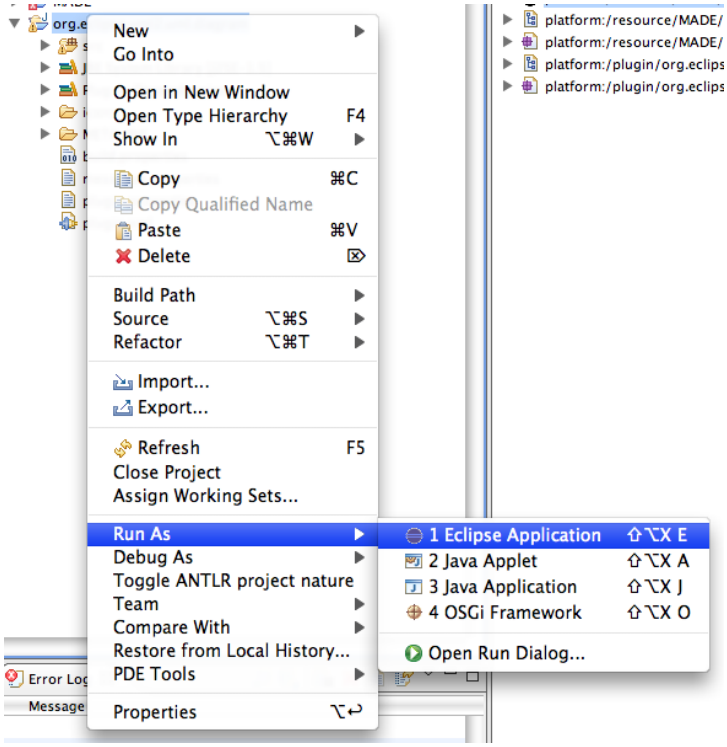

Figure 0-6 run our plug-in